Using Julia for Introductory Econometrics
1st edition

Florian Heiss

Daniel Brunner

Using Julia for Introductory Econometrics

Companion website: **http://www.UPfIE.net**

Address:
Universitätsstraße 1, Geb. 24.31.01.24
40225 Düsseldorf, Germany

Contents

List of Tables

List of Figures

Preface

An essential part of learning econometrics is the application of the methods to real-world problems and data. The practical implementation and application of econometric methods and tools helps tremendously with understanding the concepts. But learning how to use a software package also has great benefits in and of itself. Nowadays, a vast majority of our students will have to deal with some sort of data analysis in their careers. So a solid understanding of some serious data analysis software is an invaluable asset for any student of economics, business administration, and related fields.

But what software package is the right one for learning econometrics? That's a tough question. Possibly the most important aspect is that it is widely used both in and outside of academia. A large and active user community helps the software to remain up to date and increases the chances that somebody else has already solved the problem at hand. And fluency in a software package is especially valuable on the job market as well as on the job if it is popular. Another aspect for the software choice is that it is easily (and ideally freely) available to all students. This book is the latest part of a series covering the implementation of the same methods and applications using three of the best choices of software packages for these purposes.:

- "Using R for Introductory Econometrics": *R* traditionally is the most widely used software package in statistics and there is a huge community and countless packages in this area. More recently, the data wrangling and visualization capabilities using the "tidyverse" set of packages have become very popuar.
- "Using Python for Introductory Econometrics": *Python* is one of the most popular programming languages in many areas and very versatile. It has also become a *de facto* standard in areas like machine learning and AI.
- "Using Julia for Introductory Econometrics": *Julia* is the youngest of these software packages and languages. This makes it the most powerful in many aspects (like the syntax and computational speed). It also has the drawback that there are fewer packages available so far. You will see a few examples where we run *Python* code inside *Julia* to work around this problem. Chances are good that the steady development of *Julia* will make this detour unnecessary in the future.

All three books use the same structure, the same examples, and even much of the same text where it makes sense. This decision was not (only) made for laziness of the authors. It also helps readers to easily switch back and forth between the books. And if somebody worked through the *R* or *Python* book, she can easily look it up in *Julia* to achieve exactly the same results and vice versa, making it especially easy to learn all three languages. But most of all data analysis and econometrics tasks can be equally well performed in all languages. At the end, it's most important point is to get used to the workflow of *some* dedicated data analysis software package instead of not using any software or a spreadsheet program for data analysis.

The *Julia* language was released in 2012 and their authors motivated it as follows:[1]

> "We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled."

This makes *Julia* an ideal candidate for starting to learn econometrics and data analysis. As we will show in this book, learning *Julia* and the basics of econometrics are two goals that can be achieved very well together.

And *Julia* is completely free and available for all relevant operating systems. When using it in econometrics courses, students can easily download a copy to their own computers and use it at home (or their favorite cafés) to replicate examples and work on take-home assignments. This hands-on experience is essential for the understanding of the econometric models and methods. It also prepares students to conduct their own empirical analyses for their theses, research projects, and professional work.

A problem we encountered when teaching introductory econometrics classes is that the textbooks that also introduce *R*, *Python* or *Julia* do not discuss econometrics. Conversely, our favorite introductory econometrics textbooks do not cover the software. Although it is possible to combine a good econometrics textbook with an unrelated software introduction, this creates substantial hurdles because the topics and order of presentation are different, and the terminology and notation are inconsistent.

This book does not attempt to provide a self-contained discussion of econometric models and methods. Instead, it builds on the excellent and popular textbook "Introductory Econometrics" by Wooldridge (2019). It is compatible in terms of topics, organization, terminology, and notation, and is designed for a seamless transition from theory to practice.

The first chapter provides a gentle introduction to *Julia*, covers some of the topics of basic statistics and probability presented in the appendix of Wooldridge (2019), and introduces Monte Carlo simulation as an additional tool. The other chapters have the same names and cover the same material as the respective chapters in Wooldridge (2019). Assuming the reader has worked through the material discussed there, this book explains and demonstrates how to implement everything in *Julia* and replicates many textbook examples. We also open some black boxes of the built-in functions for estimation and inference by directly applying the formulas known from the textbook to reproduce the results. Some supplementary analyses provide additional intuition and insights. We want to thank Lars Grönberg and Anna Schmidt providing us with many suggestions and valuable feedback about the contents of this book.

The book is designed mainly for students of introductory econometrics who ideally use Wooldridge (2019) as their main textbook. It can also be useful for readers who are familiar with econometrics and possibly other software packages. For them, it offers an introduction to *Julia* and can be used to look up the implementation of standard econometric methods. Because we are explicitly building on Wooldridge (2019), it is useful to have a copy at hand while working through this book. All computer code used in this book can be downloaded to make it easier to replicate the results and tinker with the specifications. The companion website also provides the full text of this book for online viewing and additional material. It is located at:

`http://www.UPfIE.net`

[1] The complete text is available under `https://julialang.org/blog/2012/02/why-we-created-julia/`.

1. Introduction

Learning to use *Julia* is straightforward but not trivial. This chapter prepares us for implementing the actual econometric analyses discussed in the following chapters. First, we introduce the basics of the software system *Julia* in Section 1.1. In order to build a solid foundation we can later rely on, Chapters 1.2 through 1.4 cover the most important concepts and approaches used in *Julia* like working with objects, dealing with data, and generating graphs. Sections 1.5 through 1.7 quickly go over the most fundamental concepts in statistics and probability and show how they can be implemented in *Julia*. More advanced *Julia* topics like conditional execution, loops and functions are presented in Section 1.8. They are not really necessary for most of the material in this book. An exception is Monte Carlo simulation which is introduced in Section 1.9.

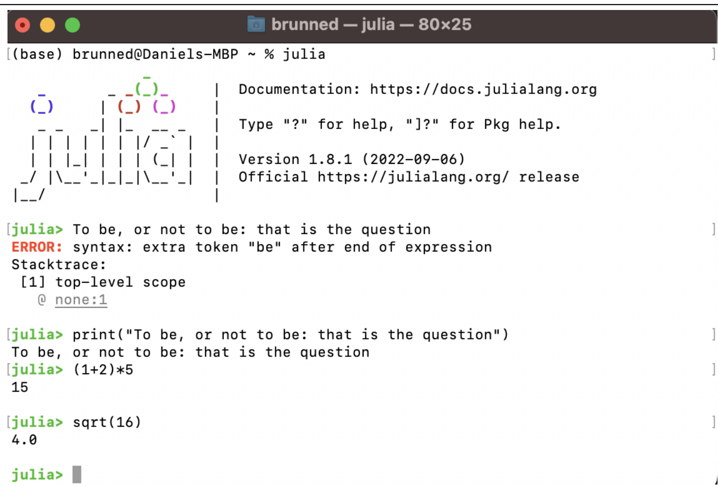
1.1. Getting Started

Before we can get going, we have to find and download the relevant software, figure out how the examples presented in this book can be easily replicated and tinkered with, and understand the most basic aspects of *Julia*. That is what this section is all about.

1.1.1. Software

Julia is a free and open source software. Its homepage is `https://julialang.org`. There, a wealth of information is available as well as the software itself. Distributions are available for Windows, Mac, and Linux systems and for all what follows, you need to install *Julia* on your platform. The examples in this book are based on the installation of version `1.8.1`.

Figure 1.1. *Julia* in the Command Line



After downloading and installing, *Julia* can be accessed by the command line interface. In Windows, run the program "Julia". In Linux or macOS you can simply open a terminal window.[1] You start *Julia* by typing **`julia`** and pressing the return key (↩). This will look similar to the screenshot in Figure 1.1. It provides some basic information on *Julia* and the installed version. Right to the "`julia>`" sign is the prompt where the user can type commands for *Julia* to evaluate. This kind of interaction with *Julia* is also called the REPL (read-eval-print loop).

We can type whatever we want here. After pressing ↩ , the line is terminated, *Julia* tries to make sense out of what is written and gives an appropriate answer. In the example shown in Figure 1.1, this was done four times. The texts we typed are shown next to the "`julia>`" text, *Julia* answers under the respective line.

Our first attempt did not work out well: We have got an error message. Unfortunately, *Julia* does not comprehend the language of Shakespeare. We will have to adjust and learn to speak *Julia*'s less poetic language. The second command shows one way to do this. Here, we provide the input to the command **`print`** in the correct syntax, so *Julia* understands that we entered text and knows what to do with it: print it out on the console. Next, we gave *Julia* simple computational tasks and got the result under the respective command. The syntax should be easy to understand – apparently, *Julia* can do simple addition and deals with the parentheses in the expected way. The same applies to the last command, because it simply calculates the square root by calling a function: **`sqrt(16)`** $= \sqrt{16} = 4$.

Julia is used by typing commands such as these. Not only Apple users may be less than impressed by the design of the user interface and the way the software is used. There are various approaches to make it more user friendly by providing a different user interface added on top of plain *Julia*. A very popular choice is Microsoft's Visual Studio Code and we use it for all what follows. You can download it for your platform under `https://visualstudio.microsoft.com/vs/`.

After installing and starting Visual Studio Code, you need to add the *Julia* extension. For that, click on the extension symbol (⊞) on the left and type `Julia` in the marketplace search box. Select *Julia* and install it.

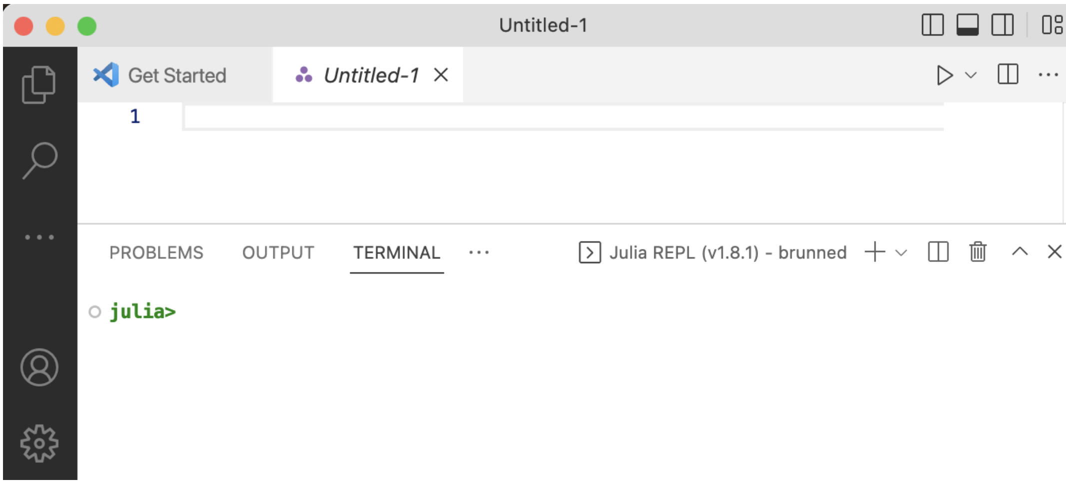
To work with *Julia*, open Visual Studio Code and click on `New File...`. You are asked what kind of file you want to create. Type `julia` and select the `Julia File` entry. A screenshot of the user interface on a Mac computer is shown in Figure 1.2 (on other systems it will look very similar). There are several sub-windows. The one on the bottom named "Terminal" looks very similar and behaves exactly the same as the command line. The usefulness of the other windows will become clear soon.

Here are a first few quick tricks for working in the terminal of Visual Studio Code:

- When starting to type a command, press ⇆ to autocomplete the command. You can try it with **`sq`** + ⇆ for the **`sqrt`** command. This only works if there is only one match. If nothing happens, you may have multiple matches and to show them press ⇆ a second time. You can try it with **`pri`** + ⇆ and get (among others) the **`print`** and **`println`** commands.
- Type `?` to enter the help mode. Then enter the name of the function you need help with and a help page for the provided command will be printed.
- With the ↑ and ↓ arrow keys, we can scroll through the previously entered commands to repeat or correct them.

[1] You may add *Julia* to the path of your platform to make this work. On a MAC, for example, you have to execute **`sudo ln -s /Applications/Julia-1.8.app/Contents/Resources/julia/bin/julia /usr/local/bin/julia`** first.

Figure 1.2. Visual Studio Code User Interface



1.1.2. *Julia* Scripts

As already seen, we will have to get used to interacting with our software using written commands. While this may seem odd to readers who do not have any experience with similar software at this point, it is actually very common for econometrics software and there are good reasons for this. An important advantage is that we can easily collect all commands we need for a project in a text file called *Julia* script.

A *Julia* script contains all commands including those for reading the raw data, data manipulation, estimation, post-estimation analyses, and the creation of graphs and tables. In a complex project, these tasks can be divided into separate *Julia* scripts. The point is that the script(s) together with the raw data generate the output used in the term paper, thesis, or research paper. We can then ask *Julia* to evaluate all or some of the commands listed in the *Julia* script at once.

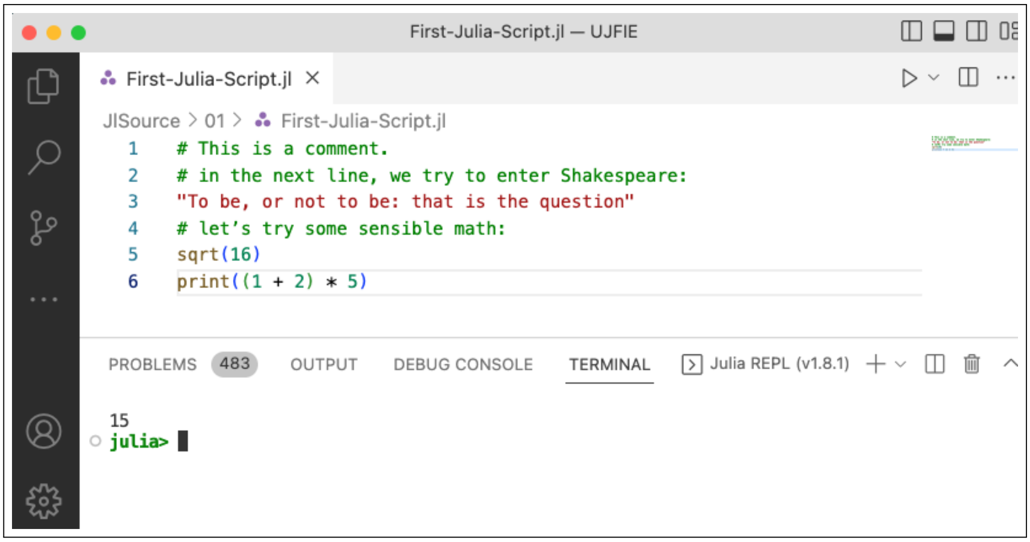
This is important since a key feature of the scientific method is reproducibility. Our thesis adviser as well as the referee in an academic peer review process or another researcher who wishes to build on our analyses must be able to fully understand where the results come from. This is easy if we can simply present our *Julia* script which has all the answers.

Working with *Julia* scripts is not only best practice from a scientific perspective, but also very convenient once we get used to it. In a nontrivial data analysis project, it is very hard to remember all the steps involved. If we manipulate the data for example by directly changing the numbers in a spreadsheet, we will never be able to keep track of everything we did. Each time we make a mistake (which is impossible to avoid), we can simply correct the command and let *Julia* start from scratch by a simple mouse click if we are using scripts. And if there is a change in the raw data set, we can simply rerun everything and get the updated tables and figures instantly.

Using *Julia* scripts is straightforward: We just write our commands into a text file and save it with a "`.jl`" extension. When using a user interface like Visual Studio Code, working with scripts is especially convenient since it is equipped with a specialized editor for script files. To use the editor for working on a new *Julia* script, click on `New File...`. You are asked what kind of file you want to create. Type `julia` and select the `Julia File` entry.

The window in the upper part of Figure 1.2 is the script editor. We can type arbitrary text, begin a new line with the return key, and navigate using the mouse or the ↑ ↓ ← → arrow keys. Our goal is not to type arbitrary text but sensible *Julia* commands. In the editor, we can also use

Figure 1.3. Executing a Script with ▷



tricks like code completion that work in the Console window as described above. A new command is generally started in a new line, but also a semicolon "`;`" can be used if we want to cram more than one command into one line – which is often not a good idea in terms of readability.

An extremely useful tool to make *Julia* scripts more readable are comments. These are lines beginning with a "**#**". These lines are not evaluated by *Julia* but can (and should) be used to structure the script and explain the steps. *Julia* scripts can be saved and opened using the `File` menu.

Figures 1.3 and 1.4 show a screenshot of Visual Studio Code with a *Julia* script saved as "First-Julia-Script.jl". It consists of six lines in total including three comments. We can send lines of code to *Julia* to be evaluated in two different ways:

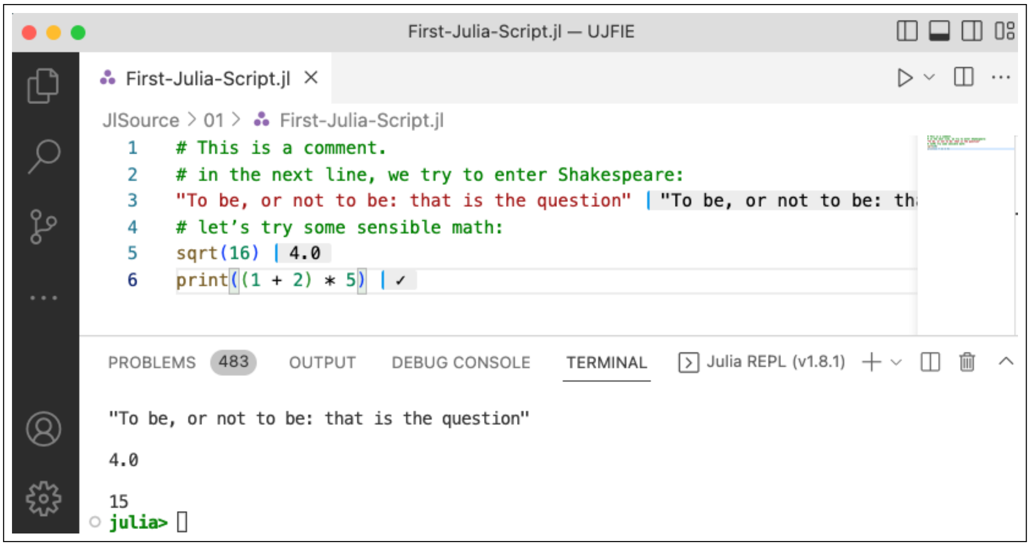
- Click ▷. The complete script is executed and only results that are explicitly printed out (by the commands **`print`** or **`println`**) show up in the "Terminal" window. The example in Figure 1.3 therefore only returns **`15`**.
- Execute *Julia* commands and scripts line by line or blockwise. The window "Terminal" shows the command you executed and the output. Press Shift ⇧ + ↩ to execute the line of the current cursor position or a highlighted block of code. As an alternative you can also right-click on the line or highlighted block or code an choose `Julia: Execute Code in REPL`. Figure 1.4 demonstrates the execution line by line.

In what follows, we will do everything using *Julia* scripts. All these scripts are available for download to make it easy and convenient to reproduce all contents in real time when reading this book. As already mentioned, the address is

`http://www.UPfIE.net`

They are also printed in Appendix IV. In the text, we will not show screenshots, but the script files printed in **bold** and (if any) *Julia*'s output in `standard font`. The latter only contains output that is explicitly printed out, just like the example in Figure 1.3. Script 1.1 (`First-Julia-Script.jl`) demonstrates the way we discuss *Julia* code in this book. To improve the readability of generated output, you can include **`\n`** as text in the **`print`** command to start a new line. For the same reason we often use **`println`**, which does this implicitly at the end of the line.

Figure 1.4. Executing a Script Line by Line



Script 1.1: `First-Julia-Script.jl`

```
# This is a comment.
# in the next line, we try to enter Shakespeare:
"To be, or not to be: that is the question"
# let's try some sensible math:
sqrt(16)
print((1 + 2) * 5)
```

Output of Script 1.1: `First-Julia-Script.jl`

```
15
```

Script 1.2 (`Julia-as-a-Calculator.jl`) is a second (and more representative) example in which *Julia* is used for simple tasks any basic calculator can do. The *Julia* script and output are:

Script 1.2: `Julia-as-a-Calculator.jl`

```
result1 = 1 + 1
print("result1 = $result1\n")

result2 = 5 * (4 - 1)^2
println("result2 = $result2")

result3 = [result1, result2]
print("result3:\n $result3")
```

Output of Script 1.2: `Julia-as-a-Calculator.jl`

```
result1 = 2
result2 = 45
result3:
 [2, 45]
```

By using the function **print("some text $variablename")** we can combine text we want to print out in combination with values of certain variables. This gives clear and readable output. We will discuss some additional hints for efficiently working with *Julia* scripts in Section 19.

1.1.3. Packages

Packages are *Julia* files that you can access to solve your problem.[2] To make use of one or multiple packages you have to import them first with the command:

```
using packagename1, packagename2, ...
```

The standard installation of *Julia* already comes with a number of built-in packages, also called the Standard Library. Script 1.3 (`Package-Statistics.jl`) demonstrates this with the **Statistics** package. All content becomes available once the package is loaded with **using Statistics**. You can access the package content directly with its objects names, or by **packagename.objectname** to avoid name conflicts with other packages. In Script 1.3 (`Package-Statistics.jl`), the functions **mean** and **var** demonstrate both ways.

Script 1.3: Package-Statistics.jl

```
using Statistics

a = [2, 6, 4, 9, 1]

a_mean = mean(a)
println("a_mean = $a_mean")

a_var = Statistics.var(a)
println("a_var = $a_var")
```

Output of Script 1.3: Package-Statistics.jl

```
a_mean = 4.4
a_var = 10.3
```

The functionality of *Julia* can also be extended relatively easily by advanced users. This is not only useful to those who are able and willing to do this, but also for a novice user who can easily make use of a wealth of extensions generated by a big and active community. Since these extensions are mostly programmed in *Julia*, everybody can check and improve the code submitted by a user, so the quality control works very well. At the time of writing there are 7,400 packages listed in the Julia General Registry on GitHub.

Downloading and installing these packages is simple with the built-in package manager **Pkg**. Execute the following *Julia* code:

```
using Pkg
Pkg.add("packagename")
```

or type **]** in the "Terminal" window to enter the package mode and execute **add packagename**.[3] In *Julia*, packages can be organized for each project individually, which is useful if your code depends on a specific version of a package.

[2] Often these *Julia* files are called modules. When several modules are linked together, they are usually referred to as a package. For the sake of simplicity, we always refer to a package in the following.

[3] For more information, see `https://pkgdocs.julialang.org/v1/`.

The following two commands are useful if you want to update a package or print out all installed packages:

```
Pkg.update("packagename")
Pkg.status()
```

As already mentioned there are thousands of packages. Here is a list of those we will use throughout this book with a short description from the respective documentation files:

- **DataFrames**: "DataFrames.jl provides a set of tools for working with tabular data in Julia."
- **Plots**: "Plots is a plotting API and toolset."
- **Distributions**: "A Julia package for probability distributions and associated functions."
- **GLM**: "Linear and generalized linear models in Julia"
- **HypothesisTests**: "HypothesisTests.jl is a Julia package that implements a wide range of hypothesis tests."
- **Econometrics**: "This package provides the functionality to estimate the following regression models: Continuous Response Models (Ordinary Least Squares, Longitudinal estimators), Nominal Response Model (Multinomial logistic regression), Ordinal Response Model"
- **RegressionTables**: "This package provides publication-quality regression tables for use with FixedEffectModels.jl and GLM.jl, as well as any package that implements the RegressionModel abstraction."
- **WooldridgeDatasets**: "This package includes all the data sets used in the Introductory Econometrics: A Modern Approach by Jeffrey Wooldridge."
- **StatsPlots**: "This package is a drop-in replacement for Plots.jl that contains many statistical recipes for concepts and types introduced in the JuliaStats organization."
- **StatsModels**: "Basic functionality for specifying, fitting, and evaluating statistical models in Julia."
- **CategoricalArrays**: "This package provides tools for working with categorical variables, both with unordered (nominal variables) and ordered categories (ordinal variables), optionally with missing values."
- **FreqTables**: "This package allows computing one- or multi-way frequency tables (a.k.a. contingency or pivot tables) from any type of vector or array."
- **CSV**: "A pure-Julia package for handling delimited text data, be it comma-delimited (csv), tab-delimited (tsv), or otherwise."
- **PyCall**: "This package provides the ability to directly call and fully interoperate with Python from the Julia language."
- **QuantileRegressions**: "Implementation of quantile regression."
- **MarketData**: "The MarketData package provides open-source financial data for research and testing."
- **Optim**: "Optim is a Julia package for optimizing functions of various kinds."
- **KernelDensity**: "Kernel density estimators for Julia."
- **BenchmarkTools**: "BenchmarkTools makes performance tracking of Julia code easy by supplying a framework for writing and running groups of benchmarks as well as comparing benchmark results."
- **Conda**: "This package allows one to use conda as a cross-platform binary provider for Julia for other Julia packages, especially to install binaries that have complicated dependencies like Python."

Of course, the installation only has to be done once per computer/user and needs an active internet connection.

We also make use of the following built-in packages, which need no installation:

- **Statistics**: “The Statistics standard library module contains basic statistics functionality.”
- **LinearAlgebra**: “In addition to (and as part of) its support for multi-dimensional arrays, Julia provides native implementations of many common and useful linear algebra operations which can be loaded with using LinearAlgebra.”
- **Random**: “Random number generation in Julia uses the Xoshiro256++ algorithm by default, with per-Task state.”
- **Dates**: “The Dates module provides two types for working with dates: Date and DateTime, representing day and millisecond precision, respectively; both are subtypes of the abstract TimeType.”
- **Pkg**: “Development repository for Julia’s package manager, shipped with Julia v1.0 and above.”

1.1.4. File Names and the Working Directory

There are several possibilities for *Julia* to interact with files. The most important ones are to import or export a data file. We might also want to save a generated figure as a graphics file or store regression tables as text, spreadsheet, or LaTeX files.

Whenever we provide *Julia* with a file name, it can include the full path on the computer. The full (i.e. “absolute”) path to a script file might be something like

```
/Users/MyJlProject/MyScript.jl
```

on a Mac or Linux system. The path is provided for Unix based operating systems using forward slashes. If you are a Windows user, you usually use back slashes instead of forward slashes, but the Unix-style will also work in *Julia*. On a Windows system, a valid path would be

```
C:/Users/MyUserName/Documents/MyJlProject/MyScript.jl
```

If we do not provide any path, *Julia* will use the current “working directory” for reading or writing files. It can be obtained by executing the command **pwd()**. To change the working directory, use the command **cd(path)**. Relative paths are interpreted relative to the current working directory. For a neat file organization, best practice is to generate a directory for each project (say `MyJlProject`) with several sub-directories (say `JlScripts`, `data`, and `figures`). At the beginning of our script, we can use **cd("/Users/MyJlProject")** and afterwards refer to a data set in the respective sub-directory as `data/MyData.csv` and to a graphics file as `figures/MyFigure.png`.[4] Note that *Julia* can handle operating system specific path formats automatically with the function **joinpath**. Since the project structure for this book is very clear and we rarely use path names, we use relative path names in Unix format when necessary.

1.1.5. Errors

Something you will experience very soon when starting to work with *Julia* (or any other similar software package) is that you will make mistakes. The main difference to learning to ride a bicycle

[4]For working with data sets, see Section 1.3.

is that when learning to use *Julia*, mistakes will not hurt. Another difference is that even people who have been using *Julia* for years make mistakes all the time.

Many mistakes will cause *Julia* to complain in the form of error messages or warnings. An important part of learning *Julia* is to roughly get an idea of what went wrong from these messages. Here is a list of frequent error messages and warnings you might get:

- **`ERROR: DomainError with xy`**: The argument **`xy`** to a function is not valid. Try **`sqrt(-1)`** to reproduce the error.
- **`ERROR: MethodError: no method matching xy`**: There is no function available, which works with the type of provided arguments. Try **`"a" + "b"`** to reproduce the error.
- **`ERROR: UndefVarError: x not defined`**: We have tried to use a variable **`x`** that isn't defined (yet). Could also be due to a typo in the variable name.
- **`ERROR: IOError: cd("pathxy"): no such file or directory (ENOENT)`**: *Julia* wasn't able to open the file. Check the working directory, path, file name.
- **`ERROR: ArgumentError: Package xyz not found in current path.`**: We mistyped the package name. Or the required package is not installed on the computer. In this case, install it as described in Section 1.1.3.

There are countless other error messages and warnings you may encounter. Some of them are easy to interpret, but others might require more investigative prowess. Often, the search engine of your choice will be helpful.

1.1.6. Other Resources

There are many useful resources helping to learn and use *Julia*. For useful books on *Julia* in general and for data science, visit `https://julialang.org/learning/books/`. Since *Julia* has a very active user community, there is also a wealth of information available for free on the internet. Here are some suggestions:

- The official *Julia* learning section (includes the documentation, tutorials and books): `https://julialang.org/learning/`
- Quantitative economic modeling with *Julia* `https://julia.quantecon.org/intro.html`
- A digital version of the book *Julia* Data Science by Storopoli, Huijzer, and Alonso (2021): `https://juliadatascience.io`
- Stack Overflow: A general discussion forum for programmers, including many *Julia* users: `https://stackoverflow.com`
- Cross Validated: Discussion forum on statistics and data analysis with an active *Julia* community: `https://stats.stackexchange.com`
- Recently, large language models like (Chat)GPT have become very powerful tools for learning a language like *Julia*. They can explain, comment, and improve given code or help to write new code from scratch.

1.2. Objects in *Julia*

Julia can work with numbers, arrays, texts, data sets, graphs, functions, and many more objects of different types. This section covers the most important ones we will frequently encounter in the remainder of this book. We will first introduce built-in objects that are available in basic *Julia*. Next, we cover objects included in the packages **`LinearAlgebra`** and **`DataFrames`** to get closer to working with real data. Finally, we look at a way to use objects from *Python* directly in *Julia*. This is useful if we need an estimator, for example, that is only implemented in *Python*, but not in *Julia*. This will happen a few times in this book, so we introduce it here.

1.2.1. Variables

We have already observed *Julia* doing some basic arithmetic calculations. From Script 1.2 (`Julia-as-a-Calculator.jl`), the general approach of *Julia* should be self-explanatory. Fundamental operators include **+**, **−**, *****, **/** for the respective arithmetic operations and parentheses **(** and **)** that work as expected.

We will often want to store results of calculations to reuse them later. For this, we can assign any result to a variable. A variable has a name and by this name you can access the assigned object. *Julia* puts few restrictions on variable names. Usually it's a good idea to start them with a small letter and include only letters, numbers, and the underscore character "_". *Julia* is case sensitive, so **x** and **X** are different variables. A special feature of *Julia* is the support of mathematical expressions in your code. Just type a LaTeX math symbol (starting with the backslash) and press . For example, if you want to use δ, just type **\delta** + .

You already saw how variables are used to reference objects in Script 1.2 (`Julia-as-a-Calculator.jl`): The content of an object is assigned by using **=**. In order to assign the result of **1 + 1** to the variable **result1**, type **result1 = 1 + 1** . A new object is created, which includes the value 2. After assigning it to **result1**, we can use **result1** in our calculations. If there was a variable with this name before, its content is overwritten.

A list of all currently defined variable names is printed by the command **varinfo**. Restart *Julia* to remove all previously defined variables from the workspace.

Up to now, we assigned results of arithmetic operations to variables. In the next sections, we will introduce more complex types of objects like texts, arrays, data sets, function definitions, and estimation results.

1.2.2. Built-in Objects in *Julia*

You might wonder what kind of objects we have dealt with so far. Script 1.4 (`Objects-in-Julia.jl`) shows how to figure this out by using the command **typeof**:

Script 1.4: `Objects-in-Julia.jl`

```
result1 = 1 + 1
# determine the type:
type_result1 = typeof(result1)
# print the result:
println("type_result1: $type_result1\n")

result2 = 2.5
type_result2 = typeof(result2)
println("type_result2: $type_result2\n")

result3 = "To be, or not to be: that is the question"
type_result3 = typeof(result3)
println("type_result3: $type_result3")
```

Table 1.1. Logical Operators

x==y	**x** is equal to **y**	**x!=y**	**x** is NOT equal to **y**
x<y	**x** is less than **y**	**!b**	NOT **b** (i.e. **true**, if **b** is **false**)
x<=y	**x** is less than or equal to **y**	**a | b**	Either **a** or **b** is **true** (or both)
x>y	**x** is greater than **y**	**a & b**	Both **a** and **b** are **true**
x>=y	**x** is greater than or equal to **y**		

Output of Script 1.4: `Objects-in-Julia.jl`

```
type_result1: Int64

type_result2: Float64

type_result3: String
```

The command **typeof** tells us that we have created integers (**Int64**), floating point numbers (**Float64**) and text objects (**String**).[5] The data type not only defines what values can be stored, but also the actions you can perform on these objects. For example, if you want to add an integer to **result3**, *Julia* will return:

```
ERROR: MethodError: no method matching +(::String, ::Int64)
```

Scalar data types like **Int64** or **Float64** contain only single values. A Boolean value, also called logical value, is another scalar data type that will become useful if you want to execute code only if one or more conditions are met. An object of type **Bool** can only take one of two values: **true** or **false**. The easiest way to generate them is to state claims which are either true or false and let *Julia* decide. Table 1.1 lists the main logical operators.

As we saw in previous examples, scalar types differ in what kind of data they can be used for:

- **Int64**: whole numbers, for example **2** or **5**
- **Float64**: numbers with a decimal point, for example **2.0** or **4.95**
- **Char**: single characters delimited by single quotes, for example **'a'** or **'b'**
- **String**: any sequence of characters delimited by double quotes, for example **"abc"**
- **Bool**: either **true** or **false**

For statistical calculations, we obviously need to work with data sets including many numbers or texts instead of scalars. The simplest way we can collect components (even components of different types) is called an **Array** in *Julia* terminology. We will first introduce the definition of this data type as well as the basics of accessing and manipulating arrays. Second, we will demonstrate functions that become useful when working on econometric problems.

In what follows, we need only one- or two-dimensional arrays, which come with special aliases in *Julia*:

- **Vector{T}** : one-dimensional array, where each component is of type **T**
- **Matrix{T}** : two-dimensional array, where each component is of type **T**

If you mix types, say a **String** and a **Int64**, the resulting object consists of components of type **Any**, an abstract type that all objects are instances of.

[5] The number 64 in **Int64** or **Float64** refers to the required memory of 64 bits to store such an object.

To define a **Vector**, we can collect different values with the following syntax:

```
test_vec = [value1, value2, ...]
```

You can access a **Vector** entry by providing the position (starting at position 1) within square brackets next to the variable name referencing the vector (see Script 1.5 (`Arrays.jl`) for an example). You can also access a range of values by using their start position i and end position j with the syntax **vectorname[i:j]**. Instead of using the position of the last entry, you could also use **end**.

A **Matrix** can be defined row by row in the following ways:

```
test_mat_a = [[v_row1_col1 v_row1_col2 ...]
    [v_row2_col1 v_row2_col2 ...]
    ...]
test_mat_b = [v_row1_col1 v_row1_col2 ...
    v_row2_col1 v_row2_col2 ...
    ...]
test_mat_c = [v_row1_col1 v_row1_col2 ... ; v_row2_col1 v_row2_col2 ... ; ...]
```

You can also provide column by column to build the **Matrix**:

```
test_mat_d = [[v_row1_col1, v_row2_col1, ...] [v_row1_col2, v_row2_col2, ...] ...]
```

Accessing entries in a **Matrix** is similar to vectors except that you need to supply two comma separated positions in square brackets: the first number gives the row, the second number gives the column. If you want to access the third row in the second column of **test_mat**, for example, you use **test_mat[3, 2]**. Matrices are important tools for econometric analyses. Appendix D of Wooldridge (2019) introduces the basic concepts of matrix algebra and we come back to it in Section 1.2.3.[6]

Script 1.5 (`Arrays.jl`) demonstrates multiple ways of accessing single or multiple entries in a vector or matrix.

Script 1.5: Arrays.jl

```
# define arrays:
testarray1D = [1, 5, 41.3, 2.0]
println("type(testarray1D): $(typeof(testarray1D))\n")

testarray2D = [4 9 8 3
    2 6 3 2
    1 1 7 4]

# same as:
#testarray2D = [4 9 8 3; 2 6 3 2; 1 1 7 4]
#testarray2D = [[4 9 8 3]
#    [ 2 6 3 2]
#    [ 1 1 7 4]]
#testarray2D = [[4, 2, 1] [9, 6, 1] [8, 3, 7] [3, 2, 4]]

# get dimensions of testarray2D:
dim = size(testarray2D)
println("dim: $dim\n")
```

[6]The strippped-down European and African textbook Wooldridge (2014) does not include the Appendix on matrix algebra.

```
# access elements by indices:
third_elem = testarray1D[3]
println("third_elem = $third_elem\n")

second_third_elem = testarray2D[2, 3] # element in 2nd row and 3rd column
println("second_third_elem = $second_third_elem\n")

second_to_third_col = testarray2D[:, 2:3] # each row in the 2nd and 3rd column
println("second_to_third_col = $second_to_third_col\n")

last_col = testarray2D[:, end] # each row in the last column
println("last_col = $last_col\n")

# access elements by array:
first_third_elem = testarray1D[[1, 3]]
println("first_third_elem: $first_third_elem\n")

# same with Boolean array:
first_third_elem2 = testarray1D[[true, false, true, false]]
println("first_third_elem2 = $first_third_elem2\n")

k = [[true false false false]
    [false false true false]
    [true false true false]]
elem_by_index = testarray2D[k] # 1st elem in 1st row, 1st elem in 3rd row...
print("elem_by_index: $elem_by_index")
```

Output of Script 1.5: `Arrays.jl`

```
type(testarray1D): Vector{Float64}

dim: (3, 4)

third_elem = 41.3

second_third_elem = 3

second_to_third_col = [9 8; 6 3; 1 7]

last_col = [3, 2, 4]

first_third_elem: [1.0, 41.3]

first_third_elem2 = [1.0, 41.3]

elem_by_index: [4, 1, 3, 7]
```

Script 1.6 (`Array-Copy.jl`) in the appendix demonstrates how to work with a copy of an array. By default you will not work on a copy when assigning it to another variable, but the underlying object. You can use **`deepcopy`** to create a copy.

There are many built-in functions that can be applied directly to arrays as one or more arguments. In case you need a function that is not already available, you can define your own function as discussed in Section 1.8.3. You call a function with the syntax:

```
functionname(argument1, argument2, ...)
```

Table 1.2. Important Functions for Vectors and Matrices

`x .+ y`	Element-wise sum of all elements in **x** and **y**
`x .- y`	Element-wise subtraction of all elements in **x** and **y**
`x ./ y`	Element-wise division of all elements in **x** and **y**
`x .* y`	Element-wise multiplication of all elements in **x** and **y**
`x .^ y`	Element-wise raising of **x** to the power of **y**
`exp.(x)`	Element-wise exponential of all elements in **x**
`sqrt.(x)`	Element-wise square root of all elements in **x**
`log.(x)`	Element-wise natural logarithm of all elements in **x**
`sum(x)`	Sum of all elements in **x**
`minimum(x)`	Minimum of all elements in **x**
`maximum(x)`	Maximum of all elements in **x**
`length(x)`	Total number of elements in **x**
`size(x)`	Rows and/ or columns of a matrix **x**
`ndims(x)`	Dimension of **x** (1 for a vector, 2 for a matrix)
`sort(x)`	Sort elements in vector **x** in ascending order
`inv(x)`	Inverse of matrix **x**
`x * y`	Matrix multiplication of matrices **x** and **y**
`transpose(x)` or `x'`	Transpose of matrix **x**

Julia functions work on a copy of the arguments by default. This means that a modification of any argument within the function has no side effect outside the function. There might be situations, where working directly on the argument instead of a copy might be more efficient. In *Julia*, the exclamation mark behind the function name, i.e. **functionname!(argument1, argument2, ...)**, implements this. The exclamation mark "warns" you that the function has side effects. / Functions are often vectorized meaning that they are applied to each of the elements in one or more arguments separately (in a very efficient way). In *Julia*, a dot behind the name of a function or before an operator implements this behaviour. For example, **exp.(vector)** and **vector .+ 2**, is the correct syntax for calculating the exponential and adding 2 to every element of a vector.

Table 1.2 lists important functions, which work on an **Array** and Script 1.7 (`Array-Functions.jl`) provides examples to see them in action.[7] Functions in the last part of Table 1.2 will be discussed in the next subsection. We will see in Section 1.5 how to obtain descriptive statistics of the data.

Script 1.7: Array-Functions.jl

```
# define arrays:
vec1 = [1, 4, 64, 36]
mat1 = [4 9 8 3
    2 6 3 2
    1 1 7 4]

# apply some functions:
sort!(vec1)
println("vec1: $vec1\n")

vec2 = sqrt.(vec1)
vec3 = vec1 .+ vec2
println("vec3: $vec3\n")
```

[7] For a complete list of mathematical functions, see `https://docs.julialang.org/en/v1/manual/mathematical-operations/`.

```
# get dimensions of mat1:
dim_mat1 = size(mat1)
println("dim_mat1: $dim_mat1")
```

Output of Script 1.7: `Array-Functions.jl`

```
vec1: [1, 4, 36, 64]

vec3: [2.0, 6.0, 42.0, 72.0]

dim_mat1: (3, 4)
```

We can also use some predefined and useful special cases of arrays. We show some of them in Script 1.8 (`Array-SpecialCases.jl`).

Script 1.8: `Array-SpecialCases.jl`

```
# initialize matrix with each element set to zero:
zero_mat = zeros(4, 3)
println("zero_mat: \n$zero_mat\n")

# initialize matrix with each element set to one:
one_mat = ones(2, 5)
println("one_mat: \n$one_mat\n")

# uninitialized matrix (filled with arbitrary nonsense elements):
empty_mat = Array{Float64}(undef, 2, 2)
println("empty_mat: \n$empty_mat")
```

Output of Script 1.8: `Array-SpecialCases.jl`

```
zero_mat:
[0.0 0.0 0.0; 0.0 0.0 0.0; 0.0 0.0 0.0; 0.0 0.0 0.0]

one_mat:
[1.0 1.0 1.0 1.0 1.0; 1.0 1.0 1.0 1.0 1.0]

empty_mat:
[0.0 2.7850472156e-314; 2.7850472156e-314 0.0]
```

A key characteristic of an **`Array`** is the order of included components. This order allows you to access its components by a position. Dictionaries (**`Dict`**) are unordered sets of components. You access components by their unique keys. A **`Dict`** can be defined in the following ways:

```
test_dict_a = Dict([("key1", object1), ("key2", object2)])
test_dict_b = Dict("key1" => object1, "key2" => object2)
```

Any component can be accessed by the **`test_dict["key"]`** syntax. Script 1.9 (`Dicts.jl`) demonstrates their definition and some basic operations.

Script 1.9: Dicts.jl

```
# define and print a dict:
var1 = ["Florian", "Daniel"]
var2 = [96, 49]
example_dict = Dict("name" => var1, "points" => var2)
println("example_dict: \n$example_dict\n")

# get data type:
type_example_dict = typeof(example_dict)
println("type_example_dict: $type_example_dict\n")

# access "points":
points_all = example_dict["points"]
println("points_all: $points_all\n")

# access "points" of Daniel:
points_daniel = example_dict["points"][2]
println("points_daniel: $points_daniel\n")

# add 4 to "points" of Daniel:
example_dict["points"][2] = example_dict["points"][2] + 4
println("example_dict: \n$example_dict\n")

# add a new component "grade":
example_dict["grade"] = [1.3, 4.0]

# delete component "points":
delete!(example_dict, "points")
print("example_dict: \n$example_dict\n")
```

Output of Script 1.9: Dicts.jl

```
example_dict:
Dict{String, Vector}("name" => ["Florian", "Daniel"], "points" => [96, 49])

type_example_dict: Dict{String, Vector}

points_all: [96, 49]

points_daniel: 49

example_dict:
Dict{String, Vector}("name" => ["Florian", "Daniel"], "points" => [96, 53])

example_dict:
Dict{String, Vector}("name" => ["Florian", "Daniel"], "grade" => [1.3, 4.0])
```

There are more important data types and functions and so far we covered only some of them. You will see them in this book only occasionally, so we will introduce them briefly:

- A **Range** stores a sequence of numbers between **start** and **stop** and can be defined by the **start:stop** syntax. The default step size of 1 can be varied by **start:stepsize:stop** or you can set the length **L** of the sequence, alternatively. The function **range** can also be used to define a **Range** with the following function calls: **range(start, stop, step=stepsize)** or **range(start, stop, length=L)**. To inspect a range, the function **collect** is useful converting the range to an array.

- A **`Tuple`** contains multiple objects with the syntax **`(object1, object2,...)`**. In a **`NamedTuple`** the components can also have names **`(name1 = object1, name2 = object2,...)`**, which makes them accessible by the **`test_tuple.name`** syntax.
- A **`Pair`** binds two objects with the syntax **`object1 => object2`** and we have already used them as an input for a **`Dict`**.
- A **`Symbol`** is defined with the colon by **`:symbolname`**. It looks similar to a string and is used if you work with variable names, for example.

Table 1.3 summarizes all relevant built-in data types plus a simple example in case you have to look them up later.

Table 1.3. *Julia* Built-in Data Types

Julia **`type`**	Data Type	Example
`Int64`	Integer	**`a = 5`**
`Float64`	Floating Point Number	**`a = 5.3`**
`String`	String	**`a = "abc"`**
`Bool`	Boolean	**`a = true`**
`Array`	Vector or Matrix	**`a = [1, 3, 1.5]`**
`Dict`	Dictionary	**`a = Dict("a" => [1, 2], "c" => [5, 3])`**
`Range`	Range	**`a = 0:4:20`**
`Tuple`	Tuple	**`a = (b=[1, 2], c=3, d="abc")`**
`Pair`	Pair	**`a = ["a", "b"] => [1, 2, 3]`**
`Symbol`	Symbol	**`a = :varname`**

1.2.3. Matrix Algebra in `LinearAlgebra.jl`

The built-in package **`LinearAlgebra`** has a powerful matrix algebra system and is loaded by:[8]

```
using LinearAlgebra
```

Basic matrix algebra includes:

- Matrix addition using the operator **`.+`** as long as the matrices have the same dimensions.
- Matrix multiplication is done with the operator **`*`** as long as the dimensions of the matrices match.
- Element-wise multiplication is implemented by the operator **`.*`**.
- Transpose of a matrix **`X`**: as **`X'`** or **`transpose(x)`**
- Inverse of a matrix **`X`**: as **`inv(X)`**

The examples in Script 1.10 (`Matrix-Operations.jl`) should help to understand the workings of these basic operations. In order to see how the OLS estimator for the multiple regression model can be calculated using matrix algebra, see Section 3.2.

[8]Note that some functionality is available even without loading the package.

Script 1.10: Matrix-Operations.jl

```
# define matrices:
mat1 = [4 9 8
        2 6 3]
mat2 = [1 5 2
        6 6 0
        4 8 3]

# use exp() and apply it to each element:
result1 = exp.(mat1)
result1_rounded = round.(result1, digits=4)
println("result1_rounded: \n$result1_rounded\n")

result2 = mat1 .+ mat2[1:2, :]
println("result2: $result2\n")

# use another function:
mat1_tr = transpose(mat1) #or simply: mat1'
println("mat1_tr: $mat1_tr\n")

# matrix algebra:
matprod = mat1 * mat2
println("matprod: $matprod")
```

Output of Script 1.10: Matrix-Operations.jl

```
result1_rounded:
[54.5982 8103.0839 2980.958; 7.3891 403.4288 20.0855]

result2: [5 14 10; 8 12 3]

mat1_tr: [4 2; 9 6; 8 3]

matprod: [90 138 32; 50 70 13]
```

1.2.4. Objects in DataFrames.jl

The package **DataFrames** builds on top of data types introduced in previous sections and allows us to work with something we will encounter almost every time we discuss an econometric application: a data frame.[9]

A data frame is a structure that collects several variables and can be thought of as a rectangular shape with the rows representing the observational units and the columns representing the variables. A data frame can contain variables of different data types (for example a numerical **Array**, an array of **String**s and so on). Before you start working with **DataFrames**, make sure that it is installed. The first line of code always is:

```
using DataFrames
```

The most important data type in **DataFrames** is **DataFrame**, which we will often simply refer to as "data frame".

Accessing elements of a variable **df** referencing an object of data type **DataFrame** can be done in multiple ways:

[9]For more information about the package, see Harris, EPRI, DuBois, White, Bouchet-Valat, Kamiński, and other contributors (2022) or https://dataframes.juliadata.org/stable/.

- Access columns/ variables by name: **df.varname1**, **df[!, [:varname1, :varname2, ...]]** or **df[!, ["varname1", "varname2", ...]]**[10]
- Access columns/ variables by integer position *i*: **df[!, i]** (also works with ranges **i:j** or vectors of integers, e.g. **[2,4,5,...]**)
- Access rows/ observations by integer position *i*: **df[i, :]** (also works with ranges **i:j** or vectors of integers, e.g. **[2,4,5,...]**)
- Access variables *and* observations by row and column integer positions *i* and *j*: **df[i, j]** (columns/ variables can also be provided by name)

Script 1.11: DataFrames.jl

```
using DataFrames

# define a DataFrame:
icecream_sales = [30, 40, 35, 130, 120, 60]
weather_coded = [0, 1, 0, 1, 1, 0]
customers = [2000, 2100, 1500, 8000, 7200, 2000]
df = DataFrame(
    icecream_sales=icecream_sales,
    weather_coded=weather_coded,
    customers=customers
)

# print the DataFrame
println("df: \n$df\n")

# access columns by variable reference:
subset1 = df[!, [:icecream_sales, :customers]]
println("subset1: \n$subset1\n")

# access second to fourth row:
subset2 = df[2:4, :]
println("subset2: \n$subset2\n")

# access rows and columns by variable integer positions:
subset3 = df[2:4, 1:2]
println("subset3: \n$subset3\n")

# access rows by variable integer positions:
subset4 = df[2:4, [:icecream_sales, :weather_coded]]
println("subset4: \n$subset4")
```

[10] **df[!, :varname1]** implements in-place modification of the variable **varname1**, while **df[:, :varname1]** works on a copy.

Output of Script 1.11: `DataFrames.jl`

```
df:
6×3 DataFrame
 Row | icecream_sales  weather_coded  customers
     | Int64           Int64          Int64
     ---------------------------------------------
   1 |             30              0       2000
   2 |             40              1       2100
   3 |             35              0       1500
   4 |            130              1       8000
   5 |            120              1       7200
   6 |             60              0       2000

subset1:
6×2 DataFrame
 Row | icecream_sales  customers
     | Int64           Int64
     ---------------------------------------------
   1 |             30       2000
   2 |             40       2100
   3 |             35       1500
   4 |            130       8000
   5 |            120       7200
   6 |             60       2000

subset2:
3×3 DataFrame
 Row | icecream_sales  weather_coded  customers
     | Int64           Int64          Int64
     ---------------------------------------------
   1 |             40              1       2100
   2 |             35              0       1500
   3 |            130              1       8000

subset3:
3×2 DataFrame
 Row | icecream_sales  weather_coded
     | Int64           Int64
     ---------------------------------------------
   1 |             40              1
   2 |             35              0
   3 |            130              1

subset4:
3×2 DataFrame
 Row | icecream_sales  weather_coded
     | Int64           Int64
     ---------------------------------------------
   1 |             40              1
   2 |             35              0
   3 |            130              1
```

Table 1.4. Important **`DataFrames`** Functions

`first(df, i)`	First `i` observations in `df`
`last(df, i)`	Last `i` observations in `df`
`describe(df)`	Print descriptive statistics in `df`
`ncol(df)`	Number of variables in `df`
`nrow(df)`	Number of observations in `df`
`names(df)`	Variable names in `df`
`df.x` or `df[!, :x]`	Access `x` in `df`
`df[i, j]`	Access variables and observations in `df` by integer position
`push!(df, row)`	Add one observation at the end of `df` in-place
`vcat(df, df2)`	Bind two data frames `df` and `df2` vertically if variable names match
`deleteat!(df, i)`	Delete row `i` in data frame `df` in-place
`sort(df, :x)`	Sort the data in `df` by variable `x` in ascending order
`subset(df, :x => ByRow(condition))`	Extract rows in `df`, which match the provided `condition` in variable `x`
`groupby(df, :x)`	Create subgroups of `df` according to `x` in a grouped data frame
`combine(gdf, :x => function)`	Apply a `function` to variable `x` in a grouped data frame `gdf`

Many economic variables of interest have a qualitative rather than quantitative interpretation. They only take a finite set of values and the outcomes don't necessarily have a numerical meaning. Instead, they represent **qualitative** information. Examples include gender, academic major, grade, marital status, state, product type or brand. In some of these examples, the order of the outcomes has a natural interpretation (such as the grades), in others, it does not (such as the state).

As a specific example, suppose we have asked our customers to rate our product on a scale between 0 (="bad"), 1 (="okay"), and 2 (="good"). We have stored the answers of our ten respondents in terms of the numbers 0,1, and 2 in an array. We could work directly with these numbers, but often, it is convenient to use a special data type from the package **`CategoricalArrays`**.[11] One advantage is that we can attach labels to the outcomes. We extend a modified example in Script 1.12 (`DataFrames-Functions.jl`), where the variable **`weather`** is coded, and demonstrate how to assign meaningful labels. The example also includes some functions from Table 1.4, i.e. adding observations or calling functions on subgroups of the data frame. The comments explain the effect of the respective action.

[11] For more information about the package, see White, Bouchet-Valat, and other contributors (2016).

Script 1.12: DataFrames-Functions.jl

```
using DataFrames, CategoricalArrays, Statistics

# define a DataFrame:
icecream_sales = [30, 40, 35, 130, 120, 60]
weather_coded = [0, 1, 0, 1, 1, 0]
customers = [2000, 2100, 1500, 8000, 7200, 2000]
df = DataFrame(
    icecream_sales=icecream_sales,
    weather_coded=weather_coded,
    customers=customers
)

# get some descriptive statistics:
descr_stats = describe(df)
println("descr_stats: \n$descr_stats\n")

# add one observation at the end in-place:
push!(df, [50, 1, 3000])
println("df: \n$df\n")

# extract observations with more than 2500 customers:
subset_df = subset(df, :customers => ByRow(>(2500)))
println("subset_df: \n$subset_df\n")

# use a CategoricalArray object to attach labels (0 = bad; 1 = good):
df.weather = recode(df[!, :weather_coded], 0 => "bad", 1 => "good")
println("df \n$df\n")

# mean sales for each weather category by
# grouping and splitting data:
grouped_data = groupby(df, :weather)
# apply the mean to icecream_sales and combine the results:
group_means = combine(grouped_data, :icecream_sales => mean)
println("group_means: \n$group_means")
```

Output of Script 1.12: DataFrames-Functions.jl

```
descr_stats:
3×7 DataFrame
 Row | variable         mean       min    median   max    nmissing  eltype
     | Symbol           Float64    Int64  Float64  Int64  Int64     DataType
     ---------------------------------------------
   1 | icecream_sales    69.1667     30     50.0    130          0  Int64
   2 | weather_coded      0.5         0      0.5      1          0  Int64
   3 | customers       3800.0      1500   2050.0   8000          0  Int64

df:
7×3 DataFrame
 Row | icecream_sales  weather_coded  customers
     | Int64           Int64          Int64
     ---------------------------------------------
   1 |             30              0       2000
   2 |             40              1       2100
   3 |             35              0       1500
   4 |            130              1       8000
   5 |            120              1       7200
   6 |             60              0       2000
   7 |             50              1       3000
```

```
subset_df:
3×3 DataFrame
 Row | icecream_sales  weather_coded  customers
     | Int64           Int64          Int64
     ---------------------------------------------
   1 |            130              1       8000
   2 |            120              1       7200
   3 |             50              1       3000

df
7×4 DataFrame
 Row | icecream_sales  weather_coded  customers  weather
     | Int64           Int64          Int64      String
     ---------------------------------------------
   1 |             30              0       2000  bad
   2 |             40              1       2100  good
   3 |             35              0       1500  bad
   4 |            130              1       8000  good
   5 |            120              1       7200  good
   6 |             60              0       2000  bad
   7 |             50              1       3000  good

group_means:
2×2 DataFrame
 Row | weather  icecream_sales_mean
     | String   Float64
     ---------------------------------------------
   1 | bad                   41.6667
   2 | good                  85.0
```

1.2.5. Using `PyCall.jl`

So far, we have relied on *Julia* and its packages to provide the functionality we need. In the few cases where this does not work, we make use of *Python*'s extensive range of modules with the **`PyCall`** package.[12] The main advantage of this package is that we can easily execute *Python* code directly within a *Julia* session.

We start by providing the basic syntax of a *Python* module installation. This is important, because **`PyCall`** uses a minimal *Python* installation, which is private to *Julia*. After installing the *Julia* package **`Conda`**, the following *Julia* code performs a module installation in *Python*:

```
using Conda
Conda.add("packagename")
```

As demonstrated in Script 1.13 (`PyCall-Simple.jl`), the syntax **`py""" pythoncode """`** can be used to execute *Python* code. Note that **`PyCall`** automatically converts many types between the two languages, so finally we deal with a *Julia* array.

[12]For more information about the package, see Johnson (2015).

Script 1.13: PyCall-Simple.jl

```
using PyCall

# define a block of Python Code:
py"""
import numpy as np

# define arrays in numpy:
mat1 = np.array([[4, 9, 8],
                 [2, 6, 3]])
mat2 = np.array([[1, 5, 2],
                 [6, 6, 0],
                 [4, 8, 3]])
# matrix algebra:
matprod_py = mat1 @ mat2
"""

# automatic type conversion from Python to Julia:
matprod = py"matprod_py"
matprod_type = typeof(matprod)
println("matprod_type: $matprod_type\n")
println("matprod: $matprod")
```

Output of Script 1.13: PyCall-Simple.jl

```
matprod_type: Matrix{Int64}

matprod: [90 138 32; 50 70 13]
```

Script 1.14 (`PyCall-Alternative.jl`) repeats the matrix multiplication, but instead of defining the input matrices in *Python*, we start with *Julia* arrays and pass it to **numpy**s **dot** function. The functionality of a *Python* module becomes available after assigning the output of **pyimport** to the *Julia* variable **np**.

Script 1.14: PyCall-Alternative.jl

```
using PyCall

# using pyimport to work with modules:
np = pyimport("numpy")

# define matrices in Julia:
mat1 = [4 9 8
        2 6 3]
mat2 = [1 5 2
        6 6 0
        4 8 3]

# ... and pass them to numpys dot function:
matprod = np.dot(mat1, mat2)
println("matprod: $matprod\n")

matprod_type = typeof(matprod)
println("matprod_type: $matprod_type")
```

Output of Script 1.14: PyCall-Alternative.jl

```
matprod: [90 138 32; 50 70 13]

matprod_type: Matrix{Int64}
```

1.3. External Data

In previous sections, we entered all of our data manually in the script files. This is a very untypical way of getting data into our computer and we will introduce more useful alternatives. These are based on the fact that many data sets are already stored somewhere else in data formats that *Julia* can handle.

1.3.1. Data Sets in the Examples

We will reproduce many of the examples from Wooldridge (2019). The companion web site of the textbook provides the sample data sets in different formats. If you have an access code that came with the textbook, they can be downloaded free of charge. The Stata data sets are also made available online at the "Instructional Stata Datasets for econometrics" collection from Boston College, maintained by Christopher F. Baum.[13]

Fortunately, we do not have to download each data set manually and import them by the functions discussed in Section 1.3.2. Instead, we can use the package **WooldridgeDatasets**.[14] First, you need to install it as explained in Section 1.1.3. By default the function **wooldridge** imports a `CSV` file and we directly convert this to a more convenient **DataFrame** as introduced in Section 1.2.4. When working with **WooldridgeDatasets**, the first line of code always is:

```
using WooldridgeDatasets, DataFrames
```

Script 1.15 (`Wooldridge.jl`) demonstrates the first lines of a typical example in this book. As you see, we are dealing with a **DataFrame** data type, so all the functions from the previous section are applicable.

Script 1.15: Wooldridge.jl

```
using WooldridgeDatasets, DataFrames

# load data:
wage1 = DataFrame(wooldridge("wage1"))

# get type:
type_wage1 = typeof(wage1)
println("type_wage1: $type_wage1\n")

# get first four observations and first eight variables:
preview_wage1 = wage1[1:4, 1:8]
println("preview_wage1: \n$preview_wage1")
```

Output of Script 1.15: Wooldridge.jl

```
type_wage1: DataFrame

preview_wage1:
4×8 DataFrame
 Row | wage     educ   exper  tenure  nonwhite  female  married  numdep
     | Float64  Int64  Int64  Int64   Int64     Int64   Int64    Int64
     ---------------------------------------------
   1 |    3.1      11      2       0         0       1        0       2
   2 |    3.24     12     22       2         0       1        1       3
   3 |    3.0      11      2       0         0       0        0       2
   4 |    6.0       8     44      28         0       0        1       0
```

[13] The address is `https://econpapers.repec.org/paper/bocbocins/`.
[14] For more information about the package, see Bhardwaj (2020).

Figure 1.5. Examples of Text Data Files

(a) `sales.txt`

```
year product1 product2 product3
2008 0 1 2
2009 3 2 4
2010 6 3 4
2011 9 5 2
2012 7 9 3
2013 8 6 2
```

(b) `sales.csv`

```
2008,0,1,2
2009,3,2,4
2010,6,3,4
2011,9,5,2
2012,7,9,3
2013,8,6,2
```

1.3.2. Import and Export of Data Files

Probably all software packages that handle data are capable of working with data stored as text files. This makes them a natural way to exchange data between different programs and users. Common file name extensions for such data files are `RAW`, `CSV` or `TXT`. Most statistics and spreadsheet programs come with their own file format to save and load data. While it is basically always possible to exchange data via text files, it might be convenient to be able to directly read or write data in the native format of some other software.

Fortunately, packages like **CSV** or **XLSX** provide the possibility for importing and exporting data from/to text files and many programs. For a `CSV` or `TXT` file, for example, two functions in the **CSV** package handle the import and export of data:[15]

- **CSV.read(path, DataFrame)**: Imports a `CSV` or `TXT` file stored at **path** as a **DataFrame**.
- **CSV.write(path, df)**: Exports a data frame **df** as a `CSV` file to the provided **path**.

Figure 1.5 shows two flavors of a raw text file containing the same data. The file `sales.txt` contains a header with the variable names. In file `sales.csv`, the columns are separated by a comma.

Text files for storing data come in different flavors, mainly differing in how the columns of the table are separated. The commands **CSV.read** and **CSV.write** provide possibilities for reading many flavors of text files which are then stored as a **DataFrame**. Script 1.16 (`Import-Export.jl`) demonstrates the import and export of the files shown in Figure 1.5. In this example, data files are stored in and exported to the folder `data`.

Script 1.16: Import-Export.jl

```
using DataFrames, CSV

# import a .CSV file with CSV.read:
df1 = CSV.read("data/sales.csv", DataFrame, delim=",",
    header=["year", "product1", "product2", "product3"])
println("df1: \n$df1\n")

# import a .txt file with CSV.read:
df2 = CSV.read("data/sales.txt", DataFrame, delim=" ")
println("df2: \n$df2\n")

# add a row to df1:
push!(df1, [2014, 10, 8, 2])
println("df1: \n$df1")
```

[15]For more information about the package, see Quinn and other contributors (2015).

```
# export with CSV.write:
CSV.write("data/sales2.csv", df1)
```

Output of Script 1.16: **Import-Export.jl**

```
df1:
6×4 DataFrame
 Row | year   product1  product2  product3
     | Int64  Int64     Int64     Int64
     ---------------------------------------------
   1 |  2008         0         1         2
   2 |  2009         3         2         4
   3 |  2010         6         3         4
   4 |  2011         9         5         2
   5 |  2012         7         9         3
   6 |  2013         8         6         2

df2:
6×4 DataFrame
 Row | year   product1  product2  product3
     | Int64  Int64     Int64     Int64
     ---------------------------------------------
   1 |  2008         0         1         2
   2 |  2009         3         2         4
   3 |  2010         6         3         4
   4 |  2011         9         5         2
   5 |  2012         7         9         3
   6 |  2013         8         6         2

df1:
7×4 DataFrame
 Row | year   product1  product2  product3
     | Int64  Int64     Int64     Int64
     ---------------------------------------------
   1 |  2008         0         1         2
   2 |  2009         3         2         4
   3 |  2010         6         3         4
   4 |  2011         9         5         2
   5 |  2012         7         9         3
   6 |  2013         8         6         2
   7 |  2014        10         8         2
```

The command **CSV.read** includes many optional arguments that can be added. Many of these arguments are detected automatically by **CSV**, but you can also specify them explicitly. The most important arguments are:

- **header**: Integer specifying the row that includes the variable names or a vector of variable names.
- **delim**: Often columns are separated by a comma, i.e. **delim=','**. Instead, an arbitrary other character can be given. **sep=';'** might be another relevant example of a separator.
- **skipto**: Integer specifying the first row to import (and skipping all previous ones).

1.3.3. Data from other Sources

The last part of this section deals with importing data from other sources than local files on your computer. We will use the package **MarketData** which makes it straightforward to query data on Yahoo Finance.[16] You have to install **MarketData** as explained in Section 1.1.3. Script 1.17 (`Import-StockData.jl`) demonstrates the workflow of importing stock data of Ford Motor Company with the function **yahoo**. All you have to do is specify the ticker symbol of the stock and start and end date with the function **DateTime**. The latter is part of the **Dates** package and creates an object of type **DateTime**, which is the native way of storing date and time data in *Julia*. To set a date, you have to provide the year, followed by the month and day. We provide more details about **DateTime** objects in Section 10.2.1.

Script 1.17: `Import-StockData.jl`

```
using DataFrames, Dates, MarketData

# download data for "F" (= Ford) and define start and end:
ticker = "F"
start_date = DateTime(2007, 12, 31)
end_date = DateTime(2017, 01, 01)

# import data as DataFrame:
F_data = DataFrame(yahoo(ticker,
    YahooOpt(period1=start_date, period2=end_date)))

preview_F_data = first(F_data, 5)
println("preview_F_data: \n$preview_F_data")
```

Output of Script 1.17: `Import-StockData.jl`

```
preview_F_data:
5×7 DataFrame
 Row | timestamp   Open     High     Low      Close    AdjClose  Volume
     | Date        Float64  Float64  Float64  Float64  Float64   Float64
     ---------------------------------------------
   1 | 2007-12-31     6.67     6.75     6.65     6.73   4.13523  2.58452e7
   2 | 2008-01-02     6.73     6.77     6.51     6.6    4.05535  3.32397e7
   3 | 2008-01-03     6.66     6.66     6.41     6.45   3.96318  4.71862e7
   4 | 2008-01-04     6.38     6.38     6.0      6.13   3.76656  5.7781e7
   5 | 2008-01-07     6.21     6.3      6.1      6.16   3.78499  4.70076e7
```

1.4. Base Graphics with `Plots.jl`

The package **Plots** is a popular and versatile tool for producing all kinds of graphs in *Julia*.[17] In this section, we discuss the overall base approach for producing graphs and the most important general types of graphs. We will only scratch the surface of **Plots**, but you will see most of the graph producing commands relevant for this book. For more information, see the package documentation. Some specific graphs used for descriptive statistics will be introduced in Section 1.5.

Before you start producing your own graphs, make sure that you install **Plots** as explained in Section 1.1.3. When working with **Plots**, the first line of code always is:

```
using Plots
```

[16]For more information about the package, see Segal (2020).
[17]For more information about the package, see Breloff (2015) or `https://docs.juliaplots.org/stable/`.

1.4.1. Basic Graphs

One very general type is a two-way graph with an abscissa and an ordinate that typically represent two variables like X and Y.

If we have data in two vectors **x** and **y**, we can easily generate scatter plots, line plots or similar two-way graphs. The command **plot** is capable of these types of graphs and we will see some of the more specialized uses later on. Script 1.18 (Graphs-Basics.jl) generates Figures 1.6(a) and (b) and demonstrates the minimum amount of code to produce a black line plot with the **plot** command with all other options on default. It also shows how to create a scatter plot with **scatter**. A legend is included by default and can be suppressed by **legend=false**. Graphs are displayed in a separate *Julia* window. The **savefig** command exports the created plot as a PDF file to the folder JlGraphs. If the folder JlGraphs does not exist yet you must create one first to execute Script 1.18 (Graphs-Basics.jl) without error.

Script 1.18: Graphs-Basics.jl

```
using Plots

# create data:
x = [1, 3, 4, 7, 8, 9]
y = [0, 3, 6, 9, 7, 8]

# plot and save:
plot(x, y, color=:black)
savefig("JlGraphs/Graphs-Basics-a.pdf")

# scatter and save:
scatter(x, y, color=:black, markershape=:dtriangle, legend=false)
savefig("JlGraphs/Graphs-Basics-b.pdf")
```

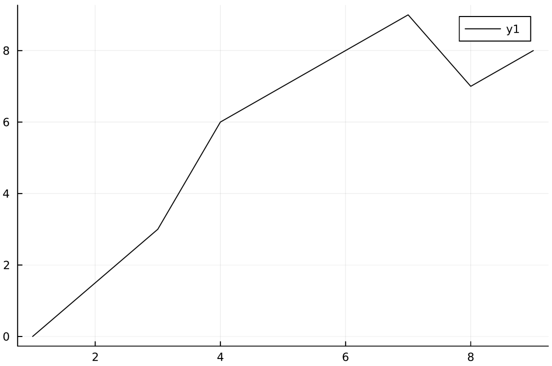
Two important arguments of the plot commands are **linestyle** and **markershape**. The argument **linestyle** takes the values **:solid** (the default), **:dash**, **:dot**, and many more. The argument **markershape** can take **:circle**, **:cross**, and many more. Some resulting plots are shown in Figure 1.6. The code is shown in the appendix in Script 1.19 (Graphs-Basics2.jl). For a complete list of arguments to customize a plot, see https://docs.juliaplots.org/latest/generated/attributes_series/.

The **plot** command can be used to create a **function plot**, i.e. function values $y = f(x)$ are plotted against x. To plot a smooth function, the first step is to generate a fine grid of x values. In Script 1.20 (Graphs-Functions.jl) we choose the **range** function and control the number of x values with **length**.[18] The following plotting of the function works exactly as in the previous example. We choose the quadratic function plotted in Figure 1.7(a) and the standard normal density (see Section 1.6) in Figure 1.7(b).

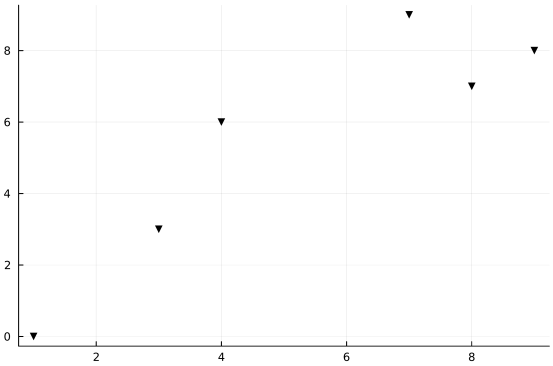
[18]The package **Distributions** will be introduced in Section 1.6.

Figure 1.6. Examples of Point and Line Plots

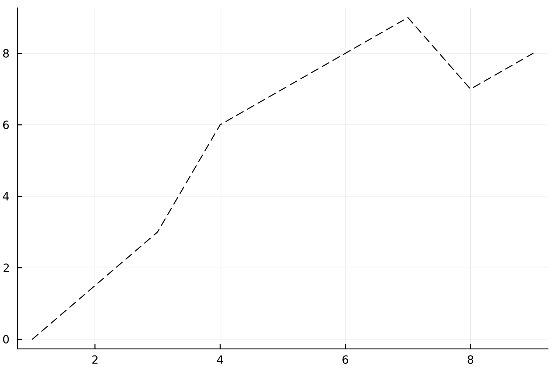
(a) see Script 1.18 (`Graphs-Basics.jl`)



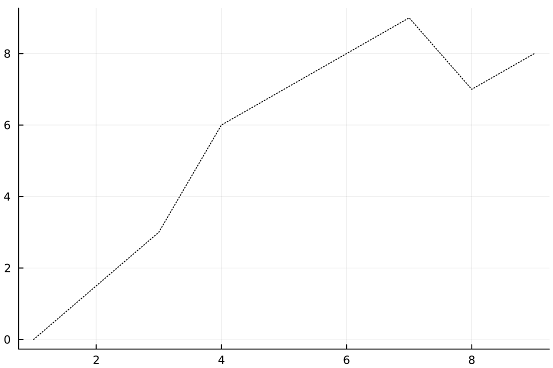
(b) see Script 1.18 (`Graphs-Basics.jl`)



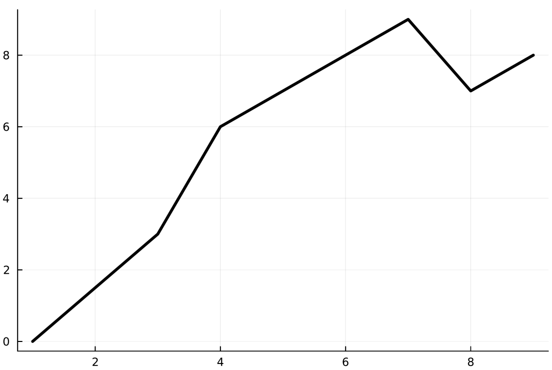
(c) **`linestyle=:dash`**



(d) **`linestyle=:dot`**



(e) **`linewidth=3`**



(f) **`markershape=:circle`**

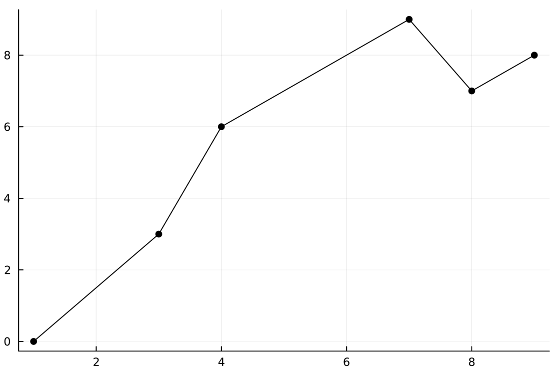
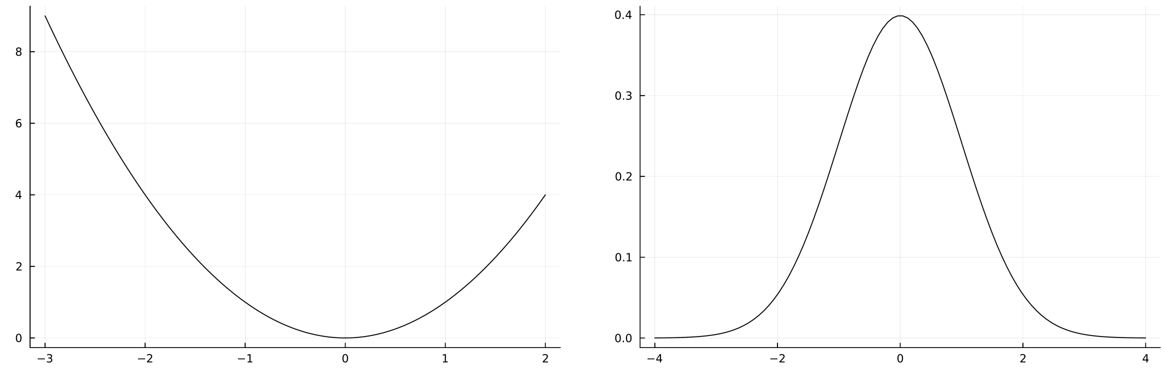


Figure 1.7. Examples of Function Plots using **`plot`**

(a) **`x1 .^ 2`** (b) **`pdf.(Normal(), x2)`**



Script 1.20: `Graphs-Functions.jl`

```
using Plots, Distributions

# support of quadratic function
# (creates an array with 100 equispaced elements from -3 to 2):
x1 = range(start=-3, stop=2, length=100)
# function values for all these values:
y1 = x1 .^ 2

# plot quadratic function:
plot(x1, y1, linestyle=:solid, color=:black, legend=false)
savefig("JlGraphs/Graphs-Functions-a.pdf")

# same for normal density:
x2 = range(-4, 4, length=100)
y2 = pdf.(Normal(), x2)

# plot normal density:
plot(x2, y2, linestyle=:solid, color=:black, legend=false)
savefig("JlGraphs/Graphs-Functions-b.pdf")
```

1.4.2. Customizing Graphs with Options

As already demonstrated in the examples, these plots can be adjusted very flexibly. A few examples:

- The width of the lines can be changed using the argument **`linewidth`**.
- The size of the marker symbols can be changed using the argument **`markersize`** (default: **`markersize=4`**).
- The transparency of a line can be changed by the argument **`linealpha`** with a number between 0 (complete transparency) and 1 (no transparency).
- The color of the lines and symbols can be changed using the argument **`color`** (also see **`linecolor`**, **`markercolor`** etc. for more flexibility). It can be specified in several ways:
  - By name: **`:blue`**, **`:green`**, **`:red`**, **`:yellow`**, **`:black`**, **`:white`**, and many more. See `http://juliagraphics.github.io/Colors.jl/stable/namedcolors/` for a complete list.

  – By RGBA values provided by (r, g, b, a) with each letter representing a number between 0 and 1, for example **`plot(x, y, color=RGBA(0.9,0.2,0.1,0.3))`**.[19] This is useful for fine-tuning colors.

You can also add more elements to change the appearance of your plot:

- A title can be added using **`title!("My Title")`**.
- The horizontal and vertical axis can be labeled using **`xlabel!("My x axis label")`** and **`ylabel!("My y axis label")`**.
- The limits of the horizontal and the vertical axis can be chosen using **`xlims!(min, max)`** and **`ylims!(min, max)`**, respectively.

For an example, see Script 1.21 (`Graphs-BuildingBlocks.jl`) and Figure 1.8.

1.4.3. Overlaying Several Plots

Often, we want to plot more than one set of variables or multiple graphical elements. This is an easy task, because each plot is added to the previous one if you use in-place modification by the **`plot!`** command.

Script 1.21 (`Graphs-BuildingBlocks.jl`) shows an example that also demonstrates some of the options from the previous paragraph. Its result is shown in Figure 1.8.[20]

Script 1.21: `Graphs-BuildingBlocks.jl`

```
using Plots, Distributions

# support for all normal densities:
x = range(-4, 4, length=100)
# get different density evaluations:
y1 = pdf.(Normal(), x)
y2 = pdf.(Normal(1, 0.5), x)
y3 = pdf.(Normal(0, 2), x)

# plot:
plot(x, y1, linestyle=:solid, color=:black, label="standard normal")
plot!(x, y2, linestyle=:dash, color=:black,
    linealpha=0.6, label="mu = 1, sigma = 0.5")
plot!(x, y3, linestyle=:dot, color=:black,
    linealpha=0.3, label="mu = 0, sigma = 2")
xlims!(-3, 4)
title!("Normal Densities")
ylabel!("phi(x)")
xlabel!("x")
savefig("JlGraphs/Graphs-BuildingBlocks.pdf")
```

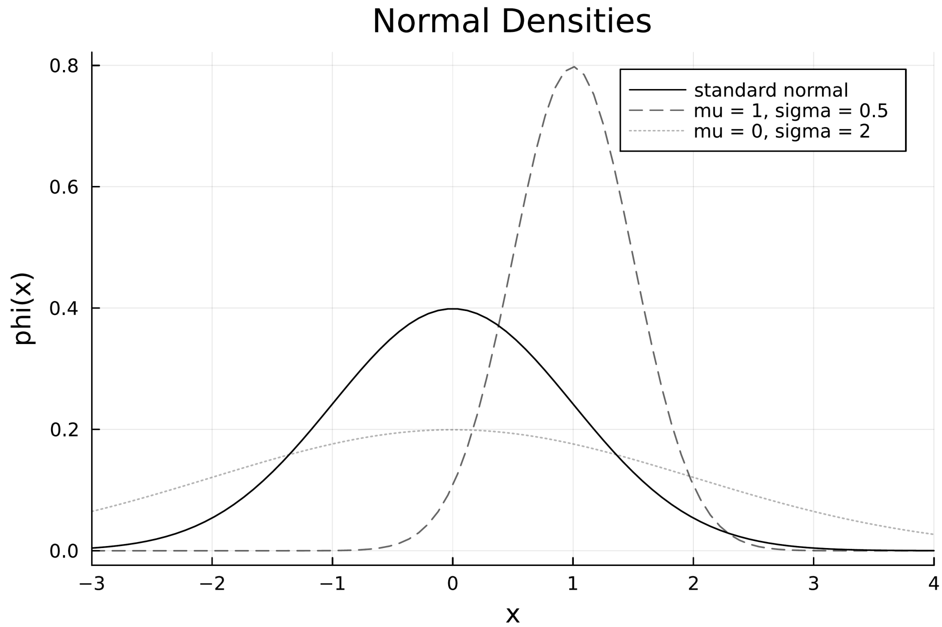
In this example, you can also see some useful commands for adding elements to an existing graph. Here are some (more) examples:

- **`hline!([y])`** adds a horizontal line at **`y`**.
- **`vline!([x])`** adds a vertical line at **`x`**.
- **`legend=position`** adds the legend at a specific **`position`**, which can be **`:topleft`**, **`:top`**, **`:topright`**, **`:left`**, **`:inside`**, **`:right`**, **`:bottomleft`**, **`:bottom`**, or **`:bottomright`**.
- **`size=(width, height)`** sets the width and height of your graph (the default is **`(600, 400)`**).

[19]The RGB color model defines colors as a mix of the components red, green, and blue. **`a`** is optional and controls for transparency.

[20]The package **`Distributions`** will be introduced in Section 1.6.

Figure 1.8. Overlayed Plots



1.4.4. Exporting to a File

By default, a graph generated in one of the ways we discussed above will be displayed in its own window. *Julia* offers the possibility to export the generated plots automatically using specific commands.

Among the different graphics formats, the `PNG` (Portable Network Graphics) format is very useful for saving plots to use them in a word processor and similar programs. For LaTeX users, `PS`, `EPS` and `SVG` are available and `PDF` is very useful. You have already seen the export syntax of the current graph in many examples:

```
savefig("filepath/filename.format")
```

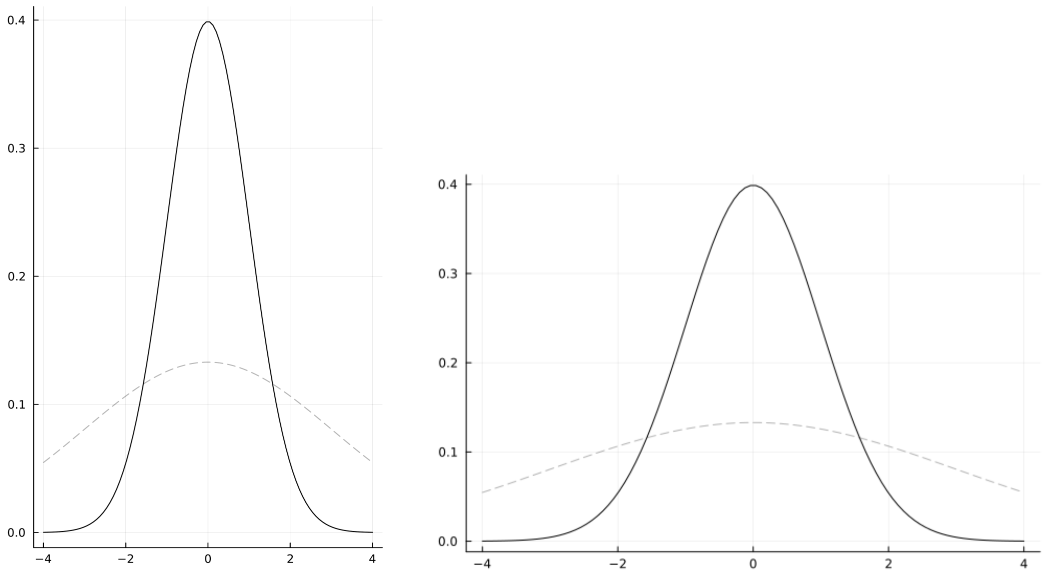
Script 1.22 (`Graphs-Export.jl`) and Figure 1.9 demonstrate the complete procedure.

Script 1.22: `Graphs-Export.jl`

```
using Plots, Distributions

# support for all normal densities:
x = range(-4, 4, length=100)
# get different density evaluations:
y1 = pdf.(Normal(), x)
y2 = pdf.(Normal(0, 3), x)

# plot (a):
plot(legend=false, size=(400, 600))
plot!(x, y1, linestyle=:solid, color=:black)
plot!(x, y2, linestyle=:dash, color=:black, linealpha=0.3)
savefig("JlGraphs/Graphs-Export-a.pdf")
```

Figure 1.9. Examples of Exported Plots

(a) **`size=(400, 600)`** (b) **`size=(600, 400)`**




```
# plot (b):
plot(legend=false, size=(600, 400))
plot!(x, y1, linestyle=:solid, color=:black)
plot!(x, y2, linestyle=:dash, color=:black, linealpha=0.3)
savefig("JlGraphs/Graphs-Export-b.png")
```

1.5. Descriptive Statistics

The *Julia* packages **`Statistics`** and **`FreqTables`** offer many commands for descriptive statistics.[21] In this section, we cover the most important ones for our purpose.

1.5.1. Discrete Distributions: Frequencies and Contingency Tables

Suppose we have a sample of the random variables X and Y stored in a data frame **`df`** as **`x`** and **`y`**, respectively. For discrete variables, the most fundamental statistics are the frequencies of outcomes. The commands **`combine(groupby(df, :x), nrow)`** or **`freqtable(df.x)`** from the package **`FreqTables`** return such a table of counts. If we are interested in the contingency table, i.e. the counts of each combination of outcomes for variables **`x`** and **`y`**, we can use **`freqtable(df.x, df.y)`**. For getting the sample *shares* instead of the *counts*, we can use the command **`proptable`**:

- The overall sample share: **`proptable(df.x, df.y)`**
- The share within **`x`** values (row percentages): **`proptable(df.x, df.y, margins=1)`**
- The share within **`y`** values (column percentages): **`proptable(df.x, df.y, margins=2)`**

[21] For more information about the package, see Bouchet-Valat (2014).

As an example, we look at the data set `affairs` in Script 1.23 (`Descr-Tables.jl`). We demonstrate the workings of the commands with two variables:

- **kids** = 1 if the respondent has at least one child
- **ratemarr** = Rating of the own marriage (1=very unhappy, ... , 5=very happy)

Script 1.23: Descr-Tables.jl

```
using WooldridgeDatasets, DataFrames, CategoricalArrays, FreqTables

affairs = DataFrame(wooldridge("affairs"))

# attach labels to kids and convert it to a categorical variable:
affairs.haskids = categorical(
    recode(affairs.kids, 0 => "no", 1 => "yes")
)

# ... and ratemarr (for example: 1 = "very unhappy", 2 = "unhappy",...):
affairs.marriage = categorical(
    recode(affairs.ratemarr,
        1 => "very unhappy",
        2 => "unhappy",
        3 => "average",
        4 => "happy",
        5 => "very happy"
    )
)

# frequency table (alphabetical order of elements):
ft_marriage = freqtable(affairs.marriage)
println("ft_marriage: \n$ft_marriage\n")

# frequency table with groupby:
ft_groupby = combine(
    groupby(affairs, :haskids),
    nrow)
println("ft_groupby: \n$ft_groupby\n")

# contingency tables with absolute and relative values:
ct_all_abs = freqtable(affairs.marriage, affairs.haskids)
println("ct_all_abs: \n$ct_all_abs\n")

ct_all_rel = proptable(affairs.marriage, affairs.haskids)
println("ct_all_rel: \n$ct_all_rel\n")

# share within "marriage" (i.e. within a row):
ct_row = proptable(affairs.marriage, affairs.haskids, margins=1)
println("ct_row: \n$ct_row\n")

# share within "haskids" (i.e. within a column):
ct_col = proptable(affairs.marriage, affairs.haskids, margins=2)
println("ct_col: \n$ct_col")
```

Output of Script 1.23: `Descr-Tables.jl`

```
ft_marriage:
5-element Named Vector{Int64}
Dim1            |
                ---------------------------------------------
"average"       |  93
"happy"         | 194
"unhappy"       |  66
"very happy"    | 232
"very unhappy"  |  16

ft_groupby:
2×2 DataFrame
 Row | haskids  nrow
     | Cat...     Int64
     ---------------------------------------------
   1 | no         171
   2 | yes        430

ct_all_abs:
5×2 Named Matrix{Int64}
   Dim1   Dim2 |  "no"  "yes"
                ---------------------------------------------
"average"       |    24     69
"happy"         |    40    154
"unhappy"       |     8     58
"very happy"    |    96    136
"very unhappy"  |     3     13

ct_all_rel:
5×2 Named Matrix{Float64}
   Dim1   Dim2 |       "no"       "yes"
                ---------------------------------------------
"average"       |  0.0399334    0.114809
"happy"         |  0.0665557     0.25624
"unhappy"       |  0.0133111   0.0965058
"very happy"    |   0.159734     0.22629
"very unhappy"  | 0.00499168   0.0216306

ct_row:
5×2 Named Matrix{Float64}
   Dim1   Dim2 |     "no"     "yes"
                ---------------------------------------------
"average"       | 0.258065  0.741935
"happy"         | 0.206186  0.793814
"unhappy"       | 0.121212  0.878788
"very happy"    | 0.413793  0.586207
"very unhappy"  |   0.1875    0.8125

ct_col:
5×2 Named Matrix{Float64}
   Dim1   Dim2 |      "no"      "yes"
                ---------------------------------------------
"average"       |  0.140351   0.160465
"happy"         |  0.233918    0.35814
"unhappy"       | 0.0467836   0.134884
"very happy"    |  0.561404   0.316279
"very unhappy"  | 0.0175439  0.0302326
```

In the *Julia* script, we first generate **categorical** versions of the two variables of interest from the coded values provided by the data set `affairs`. In this way, we can generate tables with meaningful labels instead of numbers for the outcomes, see Section 1.2.4. Then different tables are produced. Of the 601 respondents, 430 have children. Overall, 16 respondents report to be very unhappy with their marriage and 232 respondents are very happy. In the contingency table with counts, we see for example that 136 respondents are very happy and have kids.

The table reporting shares within the rows (**ct_row**) tells us that for example 81.25% of very unhappy individuals have children and only 58.6% of very happy respondents have kids. The last table reports the distribution of marriage ratings separately for people with and without kids: 56.1% of the respondents without kids are very happy, whereas only 31.6% of those with kids report to be very happy with their marriage. Before drawing any conclusions for your own family planning, please keep on studying econometrics at least until you fully appreciate the difference between correlation and causation!

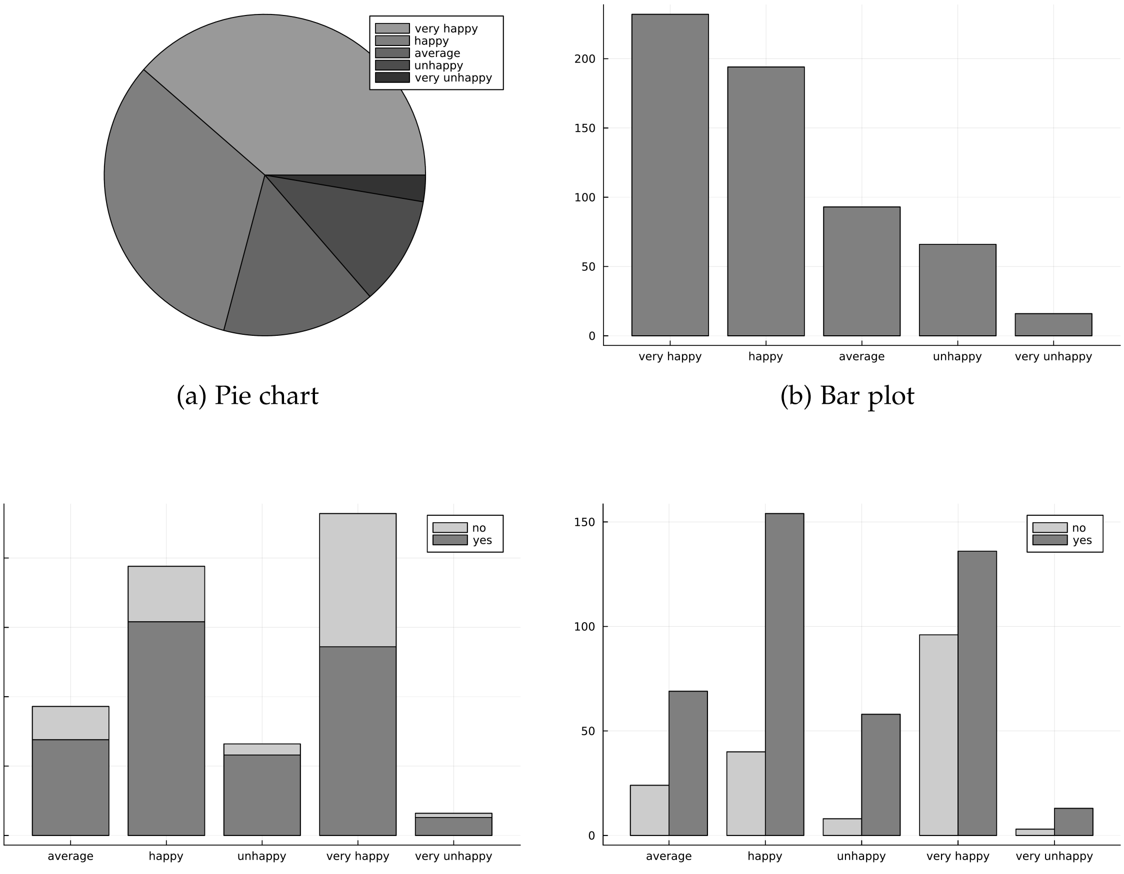
There are several ways to graphically depict the information in these tables. Script 1.24 (`Descr-Figures.jl`) demonstrates the creation of basic pie and bar charts using the commands **pie** and **bar**, respectively. It also makes use of the package **StatsPlots**, which is an add-on to **Plots** for statistical graphs.[22] These figures can of course be tweaked in many ways, see the help pages and the general discussions of graphics in Section 1.4. The best way to explore the options is to tinker with the specification and observe the results.

[22]For more information about the package, see Breloff (2016).

Script 1.24: `Descr-Figures.jl`

```
using WooldridgeDatasets, DataFrames, Plots, StatsPlots,
    FreqTables, CategoricalArrays

affairs = DataFrame(wooldridge("affairs"))

# attach labels to kids and convert it to a categorical variable:
affairs.haskids = categorical(
    recode(affairs.kids, 0 => "no", 1 => "yes")
)

# ... and ratemarr (for example: 1 = "very unhappy", 2 = "unhappy",...):
affairs.marriage = categorical(
    recode(affairs.ratemarr,
        1 => "very unhappy",
        2 => "unhappy",
        3 => "average",
        4 => "happy",
        5 => "very happy"
    )
)

# counts for all graphs:
counts_m = sort(freqtable(affairs.marriage), rev=true)
levels_counts_m = String.(collect(keys(counts_m.dicts[1])))
colors_m = [:grey60, :grey50, :grey40, :grey30, :grey20]

ct_all_abs = freqtable(affairs.marriage, affairs.haskids)
levels_counts_all = String.(collect(keys(ct_all_abs.dicts[1])))
colors_all = [:grey80 :grey50]

# pie chart (a):
pie(levels_counts_m, counts_m, color=colors_m)
savefig("JlGraphs/Descr-Pie.pdf")

# bar chart (b):
bar(levels_counts_m, counts_m, color=:grey, legend=false)
savefig("JlGraphs/Descr-Bar1.pdf")

# stacked bar plot (c):
groupedbar(ct_all_abs, bar_position=:stack,
    color=colors_all, label=["no" "yes"])
xticks!(1:5, levels_counts_all)
savefig("JlGraphs/Descr-Bar2.pdf")

# grouped bar plot (d):
groupedbar(ct_all_abs, bar_position=:dodge,
    color=colors_all, label=["no" "yes"])
xticks!(1:5, levels_counts_all)
savefig("JlGraphs/Descr-Bar3.pdf")
```

Figure 1.10. Pie and Bar Plots



(a) Pie chart

(b) Bar plot

(c) Stacked bar plot

(d) Grouped bar plot

1.5.2. Continuous Distributions: Histogram and Density

For continuous variables, every observation has a distinct value. In practice, variables which have many (but not infinitely many) different values can be treated in the same way. Since each value appears only once (or a very few times) in the data, frequency tables or bar charts are not useful. Instead, the values can be grouped into intervals. The frequency of values within these intervals can then be tabulated or depicted in a histogram.

In the *Julia* package **Plots**, the function **histogram(x, options)** assigns observations to intervals which can be manually set or automatically chosen and creates a histogram which plots values of **x** against the count or density within the corresponding bin. The most relevant options are

- **bins=...**: Set the interval boundaries:
  - no **bins** specified: let *Julia* choose number and position.
  - **bins=n** for an integer **n**: select the *number* of bins, but let *Julia* choose the position.
  - **bins=v** for a vector **v**: explicitly set the boundaries.
- **normalize=true**: do not use the count but the density on the vertical axis.

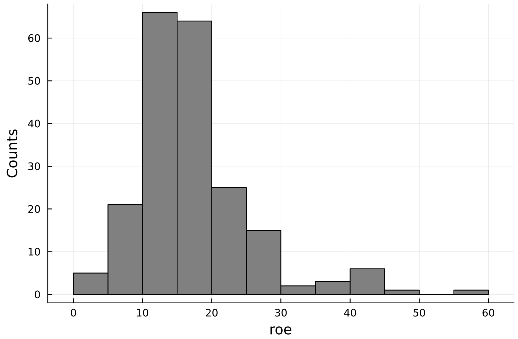
Let's look at the data set `CEOSAL1` which is described and used in Wooldridge (2019, Example 2.3). It contains information on the salary of CEOs and other information. We will try to depict the distribution of the return on equity (ROE), measured in percent. Script 1.25 (`Histogram.jl`) generates the graphs of Figure 1.11. In Sub-figure (b), the **breaks** are manually chosen and not equally spaced. Setting **normalize=true** gives the densities on the vertical axis: The sample share of observations within a bin is therefore reflected by the *area* of the respective rectangle, not the height.

Script 1.25: **Histogram.jl**

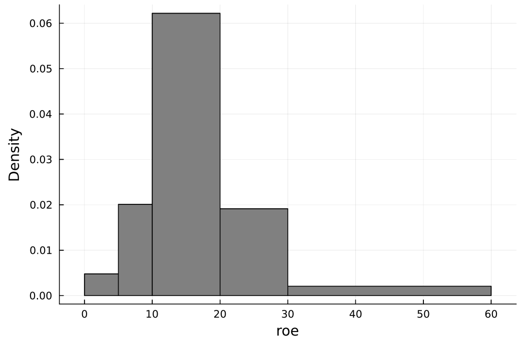
```
using WooldridgeDatasets, Plots, DataFrames

ceosal1 = DataFrame(wooldridge("ceosal1"))

# extract roe:
roe = ceosal1.roe

# histogram with counts (a):
histogram(roe, color=:grey, legend=false)
ylabel!("Counts")
xlabel!("roe")
savefig("JlGraphs/Histogram1.pdf")

# histogram with density and explicit breaks (b):
breaks = [0, 5, 10, 20, 30, 60]
histogram(roe, color=:grey,
    bins=breaks,
    normalize=true,
    legend=false)
xlabel!("roe")
ylabel!("Density")
savefig("JlGraphs/Histogram2.pdf")
```

A kernel density plot can be thought of as a more sophisticated version of a histogram. We cannot go into detail here, but an intuitive (and oversimplifying) way to think about it is this: We could create a histogram bin of a certain width, centered at an arbitrary point of x. We will do this for many points and plot these x values against the resulting densities. Here, we will not use this plot as an estimator of a population distribution but rather as a pretty alternative to a histogram for the

Figure 1.11. Histograms



(a) Histogram with counts



(b) Histogram with density and explicit breaks

descriptive characterization of the sample distribution. For details, see for example Silverman (1986). In *Julia*, generating a kernel density plot is straightforward with the package **KernelDensity**: **KernelDensity.kde(x)** will automatically choose appropriate parameters of the algorithm given the data and often produce a useful result.[23]

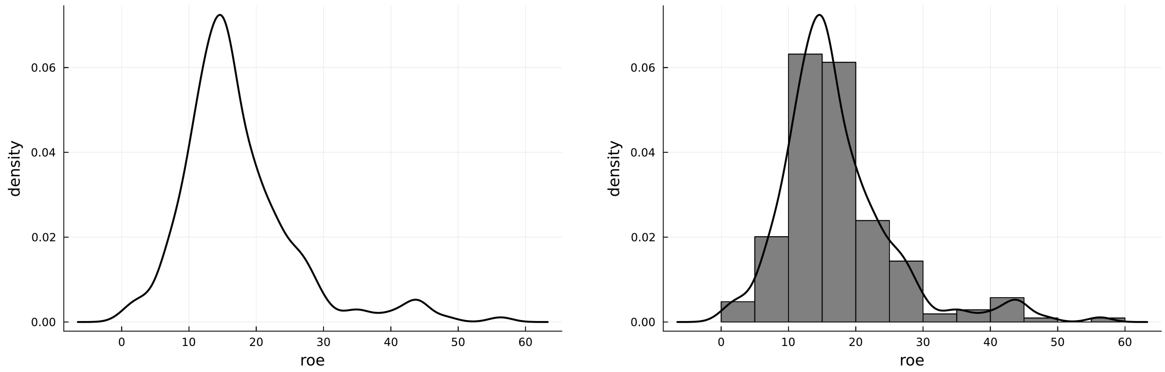
Script 1.26 (KDensity.jl) demonstrates how the result of the density estimation can be plotted with **Plots** and generates the graphs of Figure 1.12. In Sub-figure (b), a histogram is overlayed with a kernel density plot.

Script 1.26: KDensity.jl

```
using WooldridgeDatasets, DataFrames, Plots, KernelDensity

ceosal1 = DataFrame(wooldridge("ceosal1"))

# extract roe:
roe = ceosal1.roe

# estimate kernel density:
kde_est = KernelDensity.kde(roe)

# kernel density (a):
plot(kde_est.x, kde_est.density, color=:black, linewidth=2, legend=false)
ylabel!("density")
xlabel!("roe")
savefig("JlGraphs/Density1.pdf")

# kernel density with overlayed histogram (b):
histogram(roe, color="grey", normalize=true, legend=false)
plot!(kde_est.x, kde_est.density, color=:black, linewidth=2)
ylabel!("density")
xlabel!("roe")
savefig("JlGraphs/Density2.pdf")
```

[23]For more information about the package, see Byrne (2014).

Figure 1.12. Kernel Density Plots



(a) Kernel density (b) Kernel density with histogram

1.5.3. Empirical Cumulative Distribution Function (ECDF)

The ECDF is a graph of all values x of a variable against the share of observations with a value less than or equal to x. A straightforward way to plot the ECDF for our ROE variable is shown in Script 1.27 (`Descr-ECDF.jl`) and Figure 1.13. Here, the argument **`linetype=:steppre`** implements a step function.

For example, the value of the ECDF for point **`roe=15.5`** is 0.5. Half of the sample is less or equal to a ROE of 15.5%. In other words: the median ROE is 15.5%.

Script 1.27: `Descr-ECDF.jl`

```
using WooldridgeDatasets, DataFrames, Plots

ceosal1 = DataFrame(wooldridge("ceosal1"))

# extract roe:
roe = ceosal1.roe

# calculate ECDF:
x = sort(roe)
n = length(x)
y = range(start=1, stop=n) / n

# plot a step function:
plot(x, y, linetype=:steppre, color=:black, legend=false)
xlabel!("roe")
savefig("JlGraphs/ecdf.pdf")
```

1.5.4. Fundamental Statistics

The functions for calculating the most important descriptive statistics with the package **`Statistics`** are listed in Table 1.5. Script 1.28 (`Descr-Stats.jl`) demonstrates this using the `CEOSAL1` data set we already introduced in Section 1.5.2.

Figure 1.13. Empirical CDF

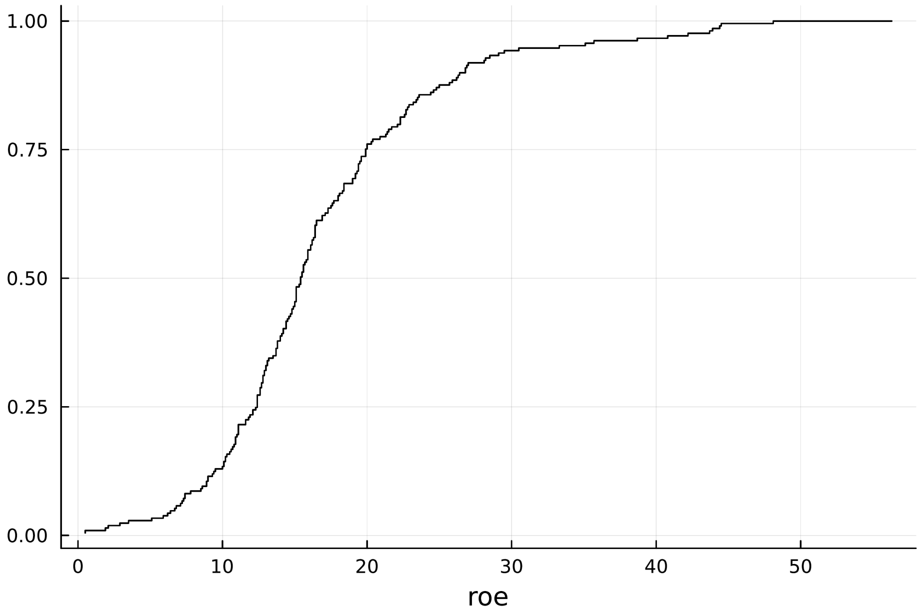


Table 1.5. `Statistics` Functions for Descriptive Statistics

`mean(x)`	Sample average $\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$
`median(x)`	Sample median
`var(x)`	Sample variance $s_x^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2$
`std(x)`	Sample standard deviation $s_x = \sqrt{s_x^2}$
`cov(x, y)`	Sample covariance $c_{xy} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})$
`cor(x, y)`	Sample correlation $r_{xy} = \frac{s_{xy}}{s_x \cdot s_y}$
`quantile(x, q)`	q quantile = 100 · q percentile, e.g. **`quantile(x, 0.5)`** = sample median

Script 1.28: `Descr-Stats.jl`

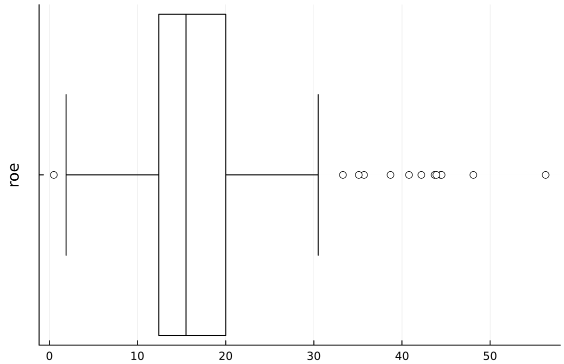
```
using WooldridgeDatasets, DataFrames, Statistics

ceosal1 = DataFrame(wooldridge("ceosal1"))

# extract roe and salary:
roe = ceosal1.roe
salary = ceosal1.salary

# sample average:
roe_mean = mean(roe)
println("roe_mean = $roe_mean\n")

# sample median:
roe_med = median(roe)
println("roe_med = $roe_med\n")

# corrected standard deviation (n-1 scaling):
roe_std = std(roe)
println("roe_st = $roe_std\n")

# correlation with roe:
roe_corr = cor(roe, salary)
println("roe_corr = $roe_corr\n")

# correlation matrix with roe:
roe_corr_mat = cor(hcat(roe, salary))
println("roe_corr_mat: \n$roe_corr_mat")
```

Output of Script 1.28: `Descr-Stats.jl`

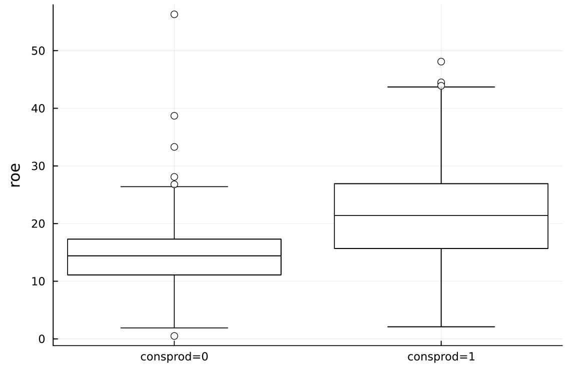
```
roe_mean = 17.18421050521175

roe_med = 15.5

roe_st = 8.518508659074904

roe_corr = 0.11484173492695977

roe_corr_mat:
[1.0 0.11484173492695976; 0.11484173492695976 1.0]
```

A box plot displays the median (the middle line), the upper and lower quartile (the box) and the extreme points graphically. Figure 1.14 shows two examples. 50% of the observations are within the interval covered by the box, 25% are above and 25% are below. The extreme points are marked by the "whiskers" and outliers are printed as separate dots. In the package **`StatsPlots`**, box plots are generated using the **`boxplot`** command. We have to supply one or more data arrays and can alter the design flexibly with numerous options as demonstrated in Script 1.29 (`Descr-Boxplot.jl`).

Figure 1.14. Box Plots



(a) Box Plot of one variable



(b) Box Plot of two variables

Script 1.29: `Descr-Boxplot.jl`

```
using WooldridgeDatasets, DataFrames, StatsPlots

ceosal1 = DataFrame(wooldridge("ceosal1"))
# extract roe and salary:
roe = ceosal1.roe
consprod = ceosal1.consprod

# plotting descriptive statistics:
boxplot(roe, orientation=:h,
    linecolor=:black, color=:white, legend=false)
yticks!([1], [""])
ylabel!("roe")
savefig("JlGraphs/Boxplot1.pdf")

# plotting descriptive statistics (logical indexing):
roe_cp0 = roe[consprod.==0]
roe_cp1 = roe[consprod.==1]
boxplot([roe_cp0, roe_cp1], linecolor=:black,
    color=:white, legend=false)
xticks!([1, 2], ["consprod=0", "consprod=1"])
ylabel!("roe")
savefig("JlGraphs/Boxplot2.pdf")
```

Figure 1.14(a) shows how to get a horizontally aligned plot and Figure 1.14(b) demonstrates how to produce multiple boxplots for two sub groups. The variable `consprod` from the data set `ceosal1` is equal to 1 if the firm is in the consumer product business and 0 otherwise. Apparently, the ROE is much higher in this industry.

1.6. Probability Distributions

Appendix B of Wooldridge (2019) introduces the concepts of random variables and their probability distributions.[24] The package **Distributions** has many functions for conveniently working with a large number of statistical distributions.[25] The commands for evaluating the probability density function (PDF) for continuous, the probability mass function (PMF) for discrete, and the cumulative distribution function (CDF) as well as the quantile function (inverse CDF) for the most relevant distributions are shown in Table 1.6. The functions are available after executing:

```
using Distributions
```

The package documentation defines the relation between the parameter set of a distribution and the function arguments in **Distributions**. We will now briefly discuss each function type.

Table 1.6. **Distributions** Functions for Statistical Distributions

Distribution	Parameters	Combine code with: • PMF/PDF: `pdf(..., x)` • CDF: `cdf(..., x)` • Quantile: `quantile(..., q)`
Discrete distributions:		
Bernoulli	p	`Bernoulli(p)`
Binomial	n, p	`Binomial(n, p)`
Hypergeometric	s, f, n	`Hypergeometric(s, f, n)`
Poisson	λ	`Poisson(λ)`
Geometric	p	`Geometric(p)`
Continuous distributions:		
Uniform	a, b	`Uniform(a, b)`
Logistic	μ, θ	`Logistic(μ, θ)`
Exponential	λ	`Exponential(1 / λ)`
Std. normal	—	`Normal()`
Normal	μ, σ	`Normal(μ, σ)`
Lognormal	μ, σ	`LogNormal(μ, σ)`
χ^2	n	`Chisq(n)`
t	n	`TDist(n)`
F	m, n	`FDist(m, n)`

1.6.1. Discrete Distributions

Discrete random variables can only take a finite (or "countably infinite") set of values. The PMF $f(x) = P(X = x)$ gives the probability that a random variable X with this distribution takes the given value x. For the most important of those distributions (Bernoulli, Binomial, Hypergeometric[26], Poisson, and Geometric[27]), Table 1.6 lists the **Distributions** functions that return the PMF for any

[24] The stripped-down textbook for Europe and Africa Wooldridge (2014) does not include this appendix. But the material is pretty standard.

[25] For more information about the package, see Bates, White, Bezanson, Karpinski, Shah, and other contributors (2012) or `https://juliastats.org/Distributions.jl/stable/`.

[26] The parameters of the distribution are defined as follows: f is the total number of unmarked balls in an urn, s is the total number of marked balls in this urn, n is the number of drawn balls and x is number of drawn marked balls.

[27] x is the total number of trials, i.e. the number of failures in a sequence of Bernoulli trials before a success occurs plus the success trial.

value x given the parameters of the respective distribution. See the package documentation, if you are interested in the formal definitions of the PMFs.

For a specific example, let X denote the number of white balls we get when drawing with replacement 10 balls from an urn that includes 20% white balls. Then X has the Binomial distribution with the parameters $n = 10$ and $p = 20\% = 0.2$. We know that the probability to get exactly $x \in \{0, 1, \ldots, 10\}$ white balls for this distribution is[28]

$$f(x) = \mathrm{P}(X = x) = \binom{n}{x} \cdot p^x \cdot (1-p)^{n-x} = \binom{10}{x} \cdot 0.2^x \cdot 0.8^{10-x} \tag{1.1}$$

For example, the probability to get exactly $x = 2$ white balls is $f(2) = \binom{10}{2} \cdot 0.2^2 \cdot 0.8^8 = 0.302$. Of course, we can let *Julia* do these calculations using basic *Julia* commands we know from Section 1.1. More conveniently, we can also use the function **`Binomial`** for the Binomial distribution:

Script 1.30: `PMF-binom.jl`

```
using Distributions

# pedestrian approach:
p1 = binomial(10, 2) * (0.2^2) * (0.8^8)
println("p1 = $p1\n")

# package function:
p2 = pdf(Binomial(10, 0.2), 2)
println("p2 = $p2")
```

Output of Script 1.30: `PMF-binom.jl`

```
p1 = 0.3019898880000002

p2 = 0.301989888
```

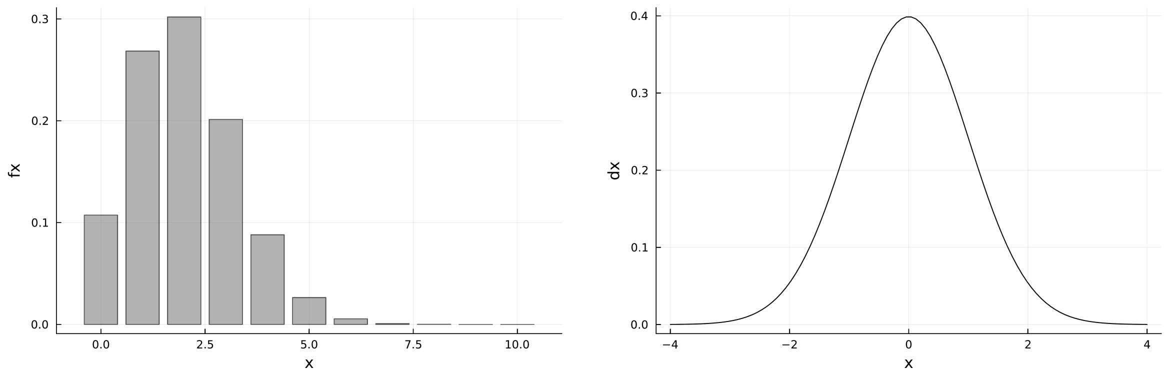
We can also give vectors as one or more arguments to **`pdf(Binomial(`**n**`,`**p**`),`**x**`)`** and receive the results as a vector. Script 1.31 (`PMF-example.jl`) evaluates the PMF for our example at all possible values for x (0 through 10). It displays a table of the probabilities and creates a bar chart of these probabilities which is shown in Figure 1.15(a). As always: feel encouraged to experiment!

Script 1.31: `PMF-example.jl`

```
using Distributions, DataFrames, Plots

# PMF for all values between 0 and 10:
x = 0:10
fx = pdf.(Binomial(10, 0.2), x)

# collect values in DataFrame:
result = DataFrame(x=x, fx=fx)
println("result: \n$result")

# plot:
bar(x, fx, color=:grey, alpha=0.6, legend=false)
xlabel!("x")
ylabel!("fx")
savefig("JlGraphs/PMF-example.pdf")
```

[28] see Wooldridge (2019, Equation (B.14))

Figure 1.15. Plots of the PMF and PDF



(a) Binomial PMF (b) Standard normal PDF

```
result:
11×2 DataFrame
 Row | x      fx
     | Int64  Float64
     ---------------------------------------------
   1 |      0  0.107374
   2 |      1  0.268435
   3 |      2  0.30199
   4 |      3  0.201327
   5 |      4  0.0880804
   6 |      5  0.0264241
   7 |      6  0.00550502
   8 |      7  0.000786432
   9 |      8  7.3728e-5
  10 |      9  4.096e-6
  11 |     10  1.024e-7
```

Output of Script 1.31: `PMF-example.jl`

1.6.2. Continuous Distributions

For continuous distributions like the uniform, logistic, exponential, normal, t, χ^2, or F distribution, the probability density functions $f(x)$ are also implemented for direct use in **`Distributions`**. These can for example be used to plot the density functions using the **`plot`** command (see Section 1.4). Figure 1.15(b) shows the famous bell-shaped PDF of the standard normal distribution and is created by Script 1.32 (`PDF-example.jl`).

Script 1.32: `PDF-example.jl`

```
using Plots, Distributions

# support of normal density:
x_range = range(-4, 4, length=100)

# PDF for all these values:
pdf_normal = pdf.(Normal(), x_range)
```

```
# plot:
plot(x_range, pdf_normal, color=:black, legend=false)
xlabel!("x")
ylabel!("dx")
savefig("JlGraphs/PDF-example.pdf")
```

1.6.3. Cumulative Distribution Function (CDF)

For all distributions, the CDF $F(x) = \mathrm{P}(X \leq x)$ represents the probability that the random variable X takes a value of *at most* x. The probability that X is between two values a and b is $\mathrm{P}(a < X \leq b) = F(b) - F(a)$. We can directly use the **Distributions** functions in Table 1.6 in combination with the function **cdf** to do these calculations as demonstrated in Script 1.33 (`CDF-example.jl`). In our example presented above, the probability that we get 3 or fewer white balls is $F(3)$ using the appropriate CDF of the Binomial distribution. It amounts to 87.9%. The probability that a standard normal random variable takes a value between -1.96 and 1.96 is 95%.

Script 1.33: `CDF-example.jl`

```
using Distributions

# binomial CDF:
p1 = cdf(Binomial(10, 0.2), 3)
println("p1 = $p1\n")

# normal CDF:
p2 = cdf(Normal(), 1.96) - cdf(Normal(), -1.96)
println("p2 = $p2")
```

Output of Script 1.33: `CDF-example.jl`

```
p1 = 0.8791261183999999

p2 = 0.950004209703559
```

Wooldridge, Example B.6: Probabilities for a Normal Random Variable

We assume $X \sim \text{Normal}(4, 9)$ and want to calculate $\mathrm{P}(2 < X \leq 6)$ as our first example. We can rewrite the problem so it is stated in terms of a standard normal distribution as shown by Wooldridge (2019): $\mathrm{P}(2 < X \leq 6) = \Phi(\frac{2}{3}) - \Phi(-\frac{2}{3})$. We can also spare ourselves the transformation and work with the non-standard normal distribution directly. Be careful that the second argument in the **Normal** command is not the variance $\sigma^2 = 9$ but the standard deviation $\sigma = 3$. The second example calculates $\mathrm{P}(|X| > 2) = \underbrace{1 - \mathrm{P}(X \leq 2)}_{\mathrm{P}(X>2)} + \mathrm{P}(X < -2)$.

Note that we get a slightly different answer in the first example than the one given in Wooldridge (2019) since we're working with the exact $\frac{2}{3}$ instead of the rounded .67.

Script 1.34: Example-B-6.jl

```
using Distributions

# first example using the transformation:
p1_1 = cdf(Normal(), 2 / 3) - cdf(Normal(), -2 / 3)
println("p1_1 = $p1_1\n")

# first example working directly with the distribution of X:
p1_2 = cdf(Normal(4, 3), 6) - cdf(Normal(4, 3), 2)
println("p1_2 = $p1_2\n")

# second example:
p2 = 1 - cdf(Normal(4, 3), 2) + cdf(Normal(4, 3), -2)
println("p2 = $p2")
```

Output of Script 1.34: Example-B-6.jl

```
p1_1 = 0.4950149249061542

p1_2 = 0.4950149249061542

p2 = 0.7702575944012563
```

The graph of the CDF is a step function for discrete distributions. For the urn example, the CDF is shown in Figure 1.16(a). The CDF of a *continuous* distribution is illustrated by the S-shaped CDF of the normal distribution as shown in Figure 1.16(b). Both figures are created by the following code:

Script 1.35: CDF-figure.jl

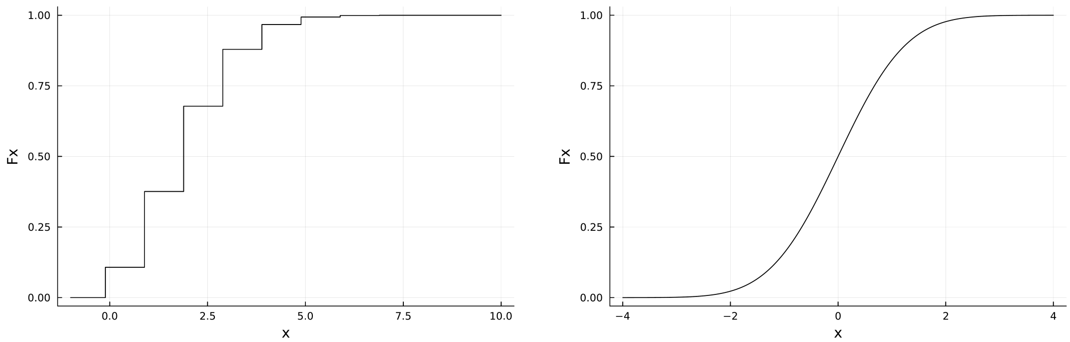
```
using Distributions, Plots

# binomial CDF:
x_binom = range(-1, 10, length=100)
cdf_binom = cdf.(Binomial(10, 0.2), x_binom)

plot(x_binom, cdf_binom, linetype=:steppre, color=:black, legend=false)
xlabel!("x")
ylabel!("Fx")
savefig("JlGraphs/CDF-figure-discrete.pdf")

# normal CDF:
x_norm = range(-4, 4, length=1000)
cdf_norm = cdf.(Normal(), x_norm)

plot(x_norm, cdf_norm, color=:black, legend=false)
xlabel!("x")
ylabel!("Fx")
savefig("JlGraphs/CDF-figure-cont.pdf")
```

Quantile Function

The q-quantile $x[q]$ of a random variable is the value for which the probability to sample a value $x \leq x[q]$ is just q. These values are important for example for calculating critical values of test statistics.

To give a simple example: Given X is standard normal, the 0.975-quantile is $x[0.975] \approx 1.96$. So the probability to sample a value less or equal to 1.96 is 97.5%:

Figure 1.16. Plots of the CDF of Discrete and Continuous RV



(a) Binomial CDF (b) Standard normal CDF

Script 1.36: `Quantile-example.jl`

```
using Distributions

q_975 = quantile(Normal(), 0.975)
println("q_975 = $q_975")
```

Output of Script 1.36: `Quantile-example.jl`

```
q_975 = 1.959963984540057б
```

1.6.4. Random Draws from Probability Distributions

It is easy to simulate random outcomes by taking a sample from a random variable with a given distribution. Strictly speaking, a deterministic machine like a computer can never produce any truly random results and we should instead refer to the generated numbers as *pseudo-random* numbers. But for our purpose, it is enough that the generated samples look, feel and behave like true random numbers and so we are a little sloppy in our terminology here. For a review of sampling and related concepts see Wooldridge (2019, Appendix C.1).

Before we make heavy use of generating random samples in Section 1.9, we introduce the mechanics here. Commands in **`Distributions`** to generate a (pseudo-) random sample are constructed by combining the command of the respective distribution (see Table 1.6) and the function **`rand`**. We could for example simulate the result of flipping a fair coin 10 times. We draw a sample of size $n = 10$ from a Bernoulli distribution with parameter $p = \frac{1}{2}$. Each of the 10 generated numbers will take the value 1 with probability $p = \frac{1}{2}$ and 0 with probability $1 - p = \frac{1}{2}$. The result behaves the same way as though we had actually flipped a coin and translated heads as 1 and tails as 0 (or vice versa). Here is the code and a sample generated by it:

Script 1.37: `Sample-Bernoulli.jl`

```
using Distributions

sample = rand(Bernoulli(0.5), 10)
println("sample: $sample")
```

Output of Script 1.37: `Sample-Bernoulli.jl`

```
sample: Bool[1, 0, 1, 0, 1, 1, 1, 1, 1, 0]
```

Translated into the coins, our sample is heads-tails-heads-tails-heads-heads-heads-heads-heads-tails. An obvious advantage of doing this in *Julia* rather than with an actual coin is that we can painlessly increase the sample size to 1,000 or 10,000,000. Taking draws from the standard normal distribution is equally simple:

Script 1.38: `Sample-Norm.jl`

```
using Distributions

sample = rand(Normal(), 6)
sample_rounded = round.(sample, digits=5)
println("sample_rounded: $sample_rounded")
```

Output of Script 1.38: `Sample-Norm.jl`

```
sample_rounded: [0.68145, -0.92263, 0.02865, 2.05688, -0.22726, 0.05151]
```

Working with computer-generated random samples creates problems for the reproducibility of the results. If you run the code above, you will get different samples. If we rerun the code, the sample will change again. We can solve this problem by making use of how the random numbers are actually generated which is, as already noted, not involving true randomness. Actually, we will always get the same sequence of numbers if we reset the random number generator to some specific state ("seed"). In *Julia*, this is can be done with the function **`Random.seed!(number)`**, where **`number`** is some arbitrary integer that defines the state but has no other meaning. The **`Random`** package is part of the standard library and needs to be loaded by **`using Random`**. If we set the seed to some arbitrary integer, take a sample, reset the seed to the same state and take another sample, both samples will be the same. Also, if I draw a sample with that seed it will be equal to the sample you draw if we both start from the same seed.

Script 1.39 (`Random-Numbers.jl`) demonstrates the workings of **`Random.seed`**.

Script 1.39: `Random-Numbers.jl`

```
using Distributions, Random

Random.seed!(12345)
# sample from a standard normal RV with sample size n=3:
sample1 = rand(Normal(), 3)
println("sample1: $sample1\n")

# a different sample from the same distribution:
sample2 = rand(Normal(), 3)
println("sample2: $sample2\n")

# set the seed of the random number generator and take two samples:
Random.seed!(54321)
sample3 = rand(Normal(), 3)
println("sample3: $sample3\n")

sample4 = rand(Normal(), 3)
println("sample4: $sample4\n")

# reset the seed to the same value to get the same samples again:
Random.seed!(54321)
sample5 = rand(Normal(), 3)
println("sample5: $sample5\n")

sample6 = rand(Normal(), 3)
println("sample6: $sample6")
```

Output of Script 1.39: `Random-Numbers.jl`

```
sample1: [2.100688289403679, -0.6969220405779567, -0.6462237060140025]

sample2: [-0.15715760672171591, -0.44322732570611395, 0.4102960875997117]

sample3: [-0.021797428733156626, -1.7563750673485294, 0.5611901926084173]

sample4: [0.8018638783150772, -0.5605955184594124, 0.6230820044249342]

sample5: [-0.021797428733156626, -1.7563750673485294, 0.5611901926084173]

sample6: [0.8018638783150772, -0.5605955184594124, 0.6230820044249342]
```

1.7. Confidence Intervals and Statistical Inference

Wooldridge (2019) provides a concise overview over basic sampling, estimation, and testing. We will touch on some of these issues below.[29]

1.7.1. Confidence Intervals

Confidence intervals (CI) are introduced in Wooldridge (2019, Appendix C.5). They are constructed to cover the true population parameter of interest with a given high probability, e.g. 95%. More clearly: For 95% of all samples, the implied CI includes the population parameter.

CI are easy to compute. For a normal population with unknown mean μ and variance σ^2, the $100(1-\alpha)\%$ confidence interval for μ is given in Wooldridge (2019, Equations C.24 and C.25):

$$\left[\bar{y} - c_{\frac{\alpha}{2}} \cdot se(\bar{y}), \quad \bar{y} + c_{\frac{\alpha}{2}} \cdot se(\bar{y})\right] \tag{1.2}$$

where $\bar{y}$ is the sample average, $se(\bar{y}) = \frac{s}{\sqrt{n}}$ is the standard error of $\bar{y}$ (with s being the sample standard deviation of y), n is the sample size and $c_{\frac{\alpha}{2}}$ the $\left(1-\frac{\alpha}{2}\right)$ quantile of the t_{n-1} distribution. To get the 95% CI ($\alpha = 5\%$), we thus need $c_{0.025}$ which is the 0.975 quantile or 97.5$^{\text{th}}$ percentile.

We already know how to calculate all these ingredients. The way of calculating the CI is used in the solution to Example C.2. In Section 1.9.3, we will calculate confidence intervals in a simulation experiment to help us understand the meaning of confidence intervals.

Wooldridge, Example C.2: Effect of Job Training Grants on Worker Productivity

We are analyzing scrap rates for firms that receive a job training grant in 1988. The scrap rates for 1987 and 1988 are printed in Wooldridge (2019, Table C.3) and are entered manually in the beginning of Script 1.40 (`Example-C-2.jl`). We are interested in the change between the years. The calculation of its average as well as the confidence interval are performed precisely as shown above. The resulting CI is the same as the one presented in Wooldridge (2019) except for rounding errors we avoid by working with the exact numbers.

[29] The stripped-down textbook for Europe and Africa Wooldridge (2014) does not include the discussion of this material.

Script 1.40: Example-C-2.jl

```
using Distributions

# manually enter raw data from Wooldridge, Table C.3:
SR87 = [10, 1, 6, 0.45, 1.25, 1.3, 1.06, 3, 8.18, 1.67,
    0.98, 1, 0.45, 5.03, 8, 9, 18, 0.28, 7, 3.97]
SR88 = [3, 1, 5, 0.5, 1.54, 1.5, 0.8, 2, 0.67, 1.17, 0.51,
    0.5, 0.61, 6.7, 4, 7, 19, 0.2, 5, 3.83]

# calculate change:
Change = SR88 .- SR87

# ingredients to CI formula:
avgCh = mean(Change)
println("avgCh = $avgCh\n")

n = length(Change)
sdCh = std(Change)
se = sdCh / sqrt(n)
println("se = $se\n")

c = quantile(TDist(n - 1), 0.975)
println("c = $c\n")

# confidence interval:
lowerCI = avgCh - c * se
println("lowerCI = $lowerCI\n")
upperCI = avgCh + c * se
println("upperCI = $upperCI")
```

Output of Script 1.40: Example-C-2.jl

```
avgCh = -1.1545

se = 0.5367992249386514

c = 2.0930240544083096

lowerCI = -2.2780336901843343

upperCI = -0.030966309815665838
```

Wooldridge, Example C.3: Race Discrimination in Hiring

We are looking into race discrimination using the data set `AUDIT`. The variable **y** represents the difference in hiring rates between black and white applicants with the identical CV. After calculating the average, sample size, standard deviation and the standard error of the sample average, Script 1.41 (`Example-C-3.jl`) calculates the value for the factor c as the 97.5 percentile of the standard normal distribution which is (very close to) 1.96. Finally, the 95% and 99% CI are reported.[30]

[30]Note that Wooldridge (2019) has a typo in the discussion of this example, therefore the numbers don't quite match for the 95% CI.

Script 1.41: `Example-C-3.jl`

```
using WooldridgeDatasets, DataFrames, Distributions

audit = DataFrame(wooldridge("audit"))
y = audit.y

# ingredients to CI formula:
avgy = mean(y)
n = length(y)
sdy = std(y)
se = sdy / sqrt(n)
c95 = quantile(Normal(), 0.975)
c99 = quantile(Normal(), 0.995)

# 95% confidence interval:
lowerCI95 = avgy - c95 * se
println("lowerCI95 = $lowerCI95\n")

upperCI95 = avgy + c95 * se
println("upperCI95 = $upperCI95\n")

# 99% confidence interval:
lowerCI99 = avgy - c99 * se
println("lowerCI99 = $lowerCI99\n")

upperCI99 = avgy + c99 * se
println("upperCI99 = $upperCI99")
```

Output of Script 1.41: `Example-C-3.jl`

```
lowerCI95 = -0.1936300609350276

upperCI95 = -0.07193010504007612

lowerCI99 = -0.2127505097677126

upperCI99 = -0.052809656207391156
```

1.7.2. *t* Tests

Hypothesis tests are covered in Wooldridge (2019, Appendix C.6). The t test statistic for testing a hypothesis about the mean μ of a normally distributed random variable Y is shown in Equation C.35. Given the null hypothesis $H_0 : \mu = \mu_0$,

$$t = \frac{\bar{y} - \mu_0}{se(\bar{y})}. \tag{1.3}$$

We already know how to calculate the ingredients from Section 1.7.1 and show to use them to perform a t test in Script 1.43 (`Example-C-5.jl`). We also compare the result to the output of the **`OneSampleTTest`** function from the package **`HypothesisTests`**, which performs an automated t test.[31]

The critical value for this test statistic depends on whether the test is one-sided or two-sided. The value needed for a two-sided test $c_{\frac{\alpha}{2}}$ was already calculated for the CI, the other values can be

[31] For more information about the package, see Kornblith and other contributors (2012) or `https://juliastats.org/HypothesisTests.jl/stable/`.

generated accordingly. The values for different degrees of freedom $n-1$ and significance levels α are listed in Wooldridge (2019, Table G.2). Script 1.42 (`Critical-Values-t.jl`) demonstrates how we can calculate our own table of critical values for the example of 19 degrees of freedom.

Script 1.42: `Critical-Values-t.jl`

```
using Distributions, DataFrames

# degrees of freedom = n-1:
df = 19

# significance levels:
alpha_one_tailed = [0.1, 0.05, 0.025, 0.01, 0.005, 0.001]
alpha_two_tailed = alpha_one_tailed * 2

# critical values & table:
CV = quantile.(TDist(df), 1 .- alpha_one_tailed)
table = DataFrame(alpha_one_tailed=alpha_one_tailed,
    alpha_two_tailed=alpha_two_tailed,
    CV=CV)
println("table: \n$table")
```

Output of Script 1.42: `Critical-Values-t.jl`

```
table:
6×3 DataFrame
 Row | alpha_one_tailed  alpha_two_tailed  CV
     | Float64           Float64           Float64
     ---------------------------------------------
   1 |            0.1               0.2    1.32773
   2 |            0.05              0.1    1.72913
   3 |            0.025             0.05   2.09302
   4 |            0.01              0.02   2.53948
   5 |            0.005             0.01   2.86093
   6 |            0.001             0.002  3.5794
```

Wooldridge, Example C.5: Race Discrimination in Hiring

We continue Example C.3 in Script 1.43 (`Example-C-5.jl`) and perform a one-sided t test of the null hypothesis $H_0 : \mu = 0$ against $H_1 : \mu < 0$ for the same sample. As the output shows, the t test statistic is equal to -4.27. This is much smaller than the negative of the critical value for any sensible significance level. Therefore, we reject $H_0 : \mu = 0$ for this one-sided test, see Wooldridge (2019, Equation C.38).

Script 1.43: `Example-C-5.jl`

```
using WooldridgeDatasets, DataFrames, Distributions, HypothesisTests

audit = DataFrame(wooldridge("audit"))
y = audit.y

# automated calculation of t statistic for H0 (mu=0):
test_auto = OneSampleTTest(y, 0)
t_auto = test_auto.t # access test statistic
p_auto = pvalue(test_auto, tail=:left)  # access one-sided p value
println("t_auto = $t_auto\n")
println("p_auto = $p_auto\n")
```

```
# manual calculation of t statistic for H0 (mu=0):
avgy = mean(y)
n = length(y)
sdy = std(y)
se = sdy / sqrt(n)
t_manual = avgy / se
println("t_manual = $t_manual\n")

# critical values for t distribution with n-1=240 d.f.:
alpha_one_tailed = [0.1, 0.05, 0.025, 0.01, 0.005, 0.001]
CV = quantile(TDist(n - 1), 1 .- alpha_one_tailed)
table = DataFrame(alpha_one_tailed=alpha_one_tailed, CV=CV)
println("table: \n$table")
```

Output of Script 1.43: `Example-C-5.jl`

```
t_auto = -4.276816348963647

p_auto = 1.3692707811129997e-5

t_manual = -4.276816348963647

table:
6×2 DataFrame
 Row | alpha_one_tailed  CV
     | Float64           Float64
    ---------------------------------------------
   1 |            0.1    1.28509
   2 |            0.05   1.65123
   3 |            0.025  1.9699
   4 |            0.01   2.34199
   5 |            0.005  2.59647
   6 |            0.001  3.12454
```

1.7.3. p Values

The p value for a test is the probability that (under the assumptions needed to derive the distribution of the test statistic) a different random sample would produce the same or an even more extreme value of the test statistic.[32] The advantage of using p values for statistical testing is that they are convenient to use. Instead of having to compare the test statistic with critical values which are implied by the significance level α, we directly compare p with α. For two-sided t tests, the formula for the p value is given in Wooldridge (2019, Equation C.42):

$$p = 2 \cdot \mathrm{P}(T_{n-1} > |t|) = 2 \cdot \left(1 - F_{t_{n-1}}(|t|)\right) \ , \tag{1.4}$$

where $F_{t_{n-1}}(\cdot)$ is the CDF of the t_{n-1} distribution which we know how to calculate from Table 1.6. Similarly, a one-sided test rejects the null hypothesis only if the value of the estimate is "too high" or "too low" relative to the null hypothesis. The p values for these types of tests are:

$$p = \begin{cases} \mathrm{P}(T_{n-1} < t) = F_{t_{n-1}}(t) & \text{for } H_1 : \mu < \mu_0 \\ \mathrm{P}(T_{n-1} > t) = 1 - F_{t_{n-1}}(t) & \text{for } H_1 : \mu > \mu_0 \end{cases} \tag{1.5}$$

Since we are working on a computer program that knows the CDF of the t distribution, calculating p values is straightforward as demonstrated in Script 1.44 (`Example-C-6.jl`). Maybe you noticed

[32]The p value is often misinterpreted. It is for example *not* the probability that the null hypothesis is true. For a discussion, see for example `https://www.nature.com/news/scientific-method-statistical-errors-1.14700`.

that the **HypothesisTests** function **pvalue** in Script 1.43 (`Example-C-5.jl`) also calculates the p value, but be aware that this function is based on two-sided t tests by default. For the one-sided t test use the argument **tail=:left** for $H_1 : \mu < \mu_0$ and **tail=:right** for $H_1 : \mu > \mu_0$.

Wooldridge, Example C.6: Effect of Job Training Grants on Worker Productivity

We continue from Example C.2 in Script 1.44 (`Example-C-6.jl`). We test $H_0 : \mu = 0$ against $H_1 : \mu < 0$. The t statistic is -2.15. The formula for the p value for this one-sided test is given in Wooldridge (2019, Equation C.41). As can be seen in the output of Script 1.44 (`Example-C-6.jl`), its value (using exact values of t) is around 0.022. For the correct p value with **HypothesisTests** use **tail=:left**, because we are dealing with a one-sided test.

Script 1.44: Example-C-6.jl

```
using Distributions, HypothesisTests

# manually enter raw data from Wooldridge, Table C.3:
SR87 = [10, 1, 6, 0.45, 1.25, 1.3, 1.06, 3, 8.18, 1.67,
        0.98, 1, 0.45, 5.03, 8, 9, 18, 0.28, 7, 3.97]
SR88 = [3, 1, 5, 0.5, 1.54, 1.5, 0.8, 2, 0.67, 1.17, 0.51,
        0.5, 0.61, 6.7, 4, 7, 19, 0.2, 5, 3.83]
Change = SR88 .- SR87

# automated calculation of t statistic for H0 (mu=0):
test_auto = OneSampleTTest(Change, 0)
t_auto = test_auto.t
p_auto = pvalue(test_auto, tail=:left)
println("t_auto = $t_auto\n")
println("p_auto = $p_auto\n")

# manual calculation of t statistic for H0 (mu=0):
avgCh = mean(Change)
n = length(Change)
sdCh = std(Change)
se = sdCh / sqrt(n)
t_manual = avgCh / se
println("t_manual = $t_manual\n")

# manual calculation of p value for H0 (mu=0):
p_manual = cdf(TDist(n - 1), t_manual)
println("p_manual = $p_manual")
```

Output of Script 1.44: Example-C-6.jl

```
t_auto = -2.1507110039734934

p_auto = 0.02229062646839212

t_manual = -2.1507110039734934

p_manual = 0.02229062646839212
```

Wooldridge, Example C.7: Race Discrimination in Hiring

In Example C.5, we found the t statistic for $H_0 : \mu = 0$ against $H_1 : \mu < 0$ to be $t = -4.276816$. The corresponding p value is calculated in Script 1.45 (`Example-C-7.jl`). The number **`1.369271e-05`** is the scientific notation for $1.369271 \cdot 10^{-5} = .00001369271$. So the p value is around 0.0014% which is much smaller than any reasonable significance level. By construction, we draw the same conclusion as when we compare the t statistic with the critical value in Example C.5. We reject the null hypothesis that there is no discrimination.

Script 1.45: `Example-C-7.jl`

```
using WooldridgeDatasets, DataFrames, Distributions, HypothesisTests

audit = DataFrame(wooldridge("audit"))
y = audit.y

# automated calculation of t statistic for H0 (mu=0):
test_auto = OneSampleTTest(y, 0)
t_auto = test_auto.t
p_auto = pvalue(test_auto, tail=:left)
println("t_auto = $t_auto\n")
println("p_auto = $p_auto\n")

# manual calculation of t statistic for H0 (mu=0):
avgy = mean(y)
n = length(y)
sdy = std(y)
se = sdy / sqrt(n)
t_manual = avgy / se
println("t_manual = $t_manual\n")

# manual calculation of p value for H0 (mu=0):
p_manual = cdf(TDist(n - 1), t_manual)
println("p_manual = $p_manual")
```

Output of Script 1.45: `Example-C-7.jl`

```
t_auto = -4.276816348963647

p_auto = 1.3692707811129997e-5

t_manual = -4.276816348963647

p_manual = 1.3692707811129997e-5
```

1.8. Advanced *Julia*

The material covered in this section is not necessary for most of what we will do in the remainder of this book, so it can be skipped. However, it is important enough to justify an own section in this chapter. We will only scratch the surface, though. For more details, see the references in Section 1.1.6.

1.8.1. Conditional Execution

We might want some parts of our code to be executed only under certain conditions. Like most other programming languages, this can be achieved with an `if else` statement with the following syntax:

```
if condition
    expression1
else
    expression2
end
```

The `condition` has to be a single logical value (`true` or `false`). If it is `true`, then `expression1` is executed, otherwise `expression2` which can also be omitted. A simple example would be

```
if p <= 0.05
    print("reject H0!")
else
    print("don’t reject H0!")
end
```

Depending on the value of the numeric scalar `p`, the respective test decision is printed. The command `ifelse` implements the same functionality and in some cases it produces more compact code and higher performance. The code `ifelse(condition, expression1, expression2)` is equivalent to the `if else` statement above. You can also use it in a vectorized form with `ifelse.` by providing a vector of conditions. The following example anticipates the example from Script 1.46 (`Adv-Loops.jl`), but without the explicit formulation of a loop:

```
seq = [1, 2, 3, 4, 5, 6]
ifelse.(seq .< 4, seq .^ 3, seq .^ 2)
```

1.8.2. Loops

For repeatedly executing an expression, different kinds of loops are available. In this book, we will use them for Monte Carlo analyses introduced in Section 1.9. For our purposes, the `for` loop is well suited. The correct syntax is:

```
for x in sequence
    [some commands]
end
```

The loop variable `x` will take the value of each element of `sequence`, one after another. For each of these elements, `[some commands]` are executed. Often, `sequence` will be an array like `[1, 2, 3]`.

A nonsense example which combines **`for`** loops with an **`if`** statement is given in Script 1.46 (`Adv-Loops.jl`). The reader is encouraged to first form expectations about the output this will generate and then compare them with the actual results.

Script 1.46: `Adv-Loops.jl`
```
seq = [1, 2, 3, 4, 5, 6]
for i in seq
    if i < 4
        println(i^3)
    else
        println(i^2)
    end
end
```

Output of Script 1.46: `Adv-Loops.jl`
```
1
8
27
16
25
36
```

Instead of iterating over a sequence you can also iterate over an index of a sequence and use the index to reference other objects. Another way of generating such a sequence of indices uses the function **`eachindex`**, which is demonstrated in Script 1.47 (`Adv-Loops2.jl`) by doing the same as Script 1.46 (`Adv-Loops.jl`).

Script 1.47: `Adv-Loops2.jl`
```
seq = [1, 2, 3, 4, 5, 6]

for i in eachindex(seq)
    if seq[i] < 4
        println(seq[i]^3)
    else
        println(seq[i]^2)
    end
end
```

Output of Script 1.47: `Adv-Loops2.jl`
```
1
8
27
16
25
36
```

If you want to execute expressions as long as a given condition is **`true`**, *Julia* offers the **`while`** loop, but we will not present it here.

1.8.3. Functions

A function is a block of code that is executed if the function is called. You can provide additional data to the function in form of arguments. There are many pre-defined functions and packages providing even more functions to expand the capabilities of *Julia*. We're now ready to define our own little function.

The command **`function newfunc(arg1, arg2, ...)`** defines a new function **`newfunc`** which accepts the arguments **`arg1`**, **`arg2`**,.... The function definition follows in arbitrarily many lines of code and is ended with **`end`**. Within the function definition, the command **`return stuff`** means that **`stuff`** is to be returned as a result of the function call. For example, we can define the function **`mysqrt`** that expects one argument internally named **`x`**. Script 1.48 (`Adv-Functions.jl`) shows how to define and call the function **`mysqrt`**.

Script 1.48: `Adv-Functions.jl`

```
# define function:
function mysqrt(x)
    if x >= 0
        result = x^0.5
    else
        result = "You fool!"
    end
    return result
end

# call function and save result:
result1 = mysqrt(4)
println("result1 = $result1\n")

result2 = mysqrt(-1.5)
println("result2 = $result2")
```

Output of Script 1.48: `Adv-Functions.jl`

```
result1 = 2.0

result2 = You fool!
```

By default, variables defined within a function do not interfere with variables outside the function. This scoping rule can be made more explicit by adding **`local`** in front of a functions variable, e.g. **`local result`**. In some cases you may need to access variables outside the function, which is implemented by using **`global`** in front of the variable of a function.

If functions are defined as in Script 1.48 (`Adv-Functions.jl`), arguments are passed to the function by position. In Script 1.48 (`Adv-Functions.jl`) it is clear that any provided input to the function must be the argument **`x`**. In the case of multiple arguments the order of provided inputs matters: the first piece of input is related to the first argument in the function definition, the second piece of input is related to the second argument in the function definition, etc. .

If you need many arguments in your function (remember the **`plot`** function, for example) providing arguments by name is an alternative easier to read. In *Julia*, this kind of argument is called a keyword argument. Note that the order of provided keyword arguments does not matter. When defining a function in *Julia*, you must specify which arguments are chosen by position and name. These both sets of arguments are separated by a semicolon in the function definition:

```
function newfunc(pos_arg1, pos_arg2, ... ; kwd_arg1, kwd_arg2, ...)
```

For an example, see Script 1.49 (`Adv-Functions-MultArg.jl`).

Script 1.49: **Adv-Functions-MultArg.jl**

```
# define function (only positional arguments):
function mysqrt_pos(x, add)
    if x >= 0
        result = x^0.5 + add
    else
        result = "You fool!"
    end
    return result
end

# define function ("x" as positional and "add" as keyword argument):
function mysqrt_kwd(x; add)
    if x >= 0
        result = x^0.5 + add
    else
        result = "You fool!"
    end
    return result
end

# call functions:
result1 = mysqrt_pos(4, 3)
println("result1 = $result1")
# mysqrt_pos(4, add = 3) is not valid

result2 = mysqrt_kwd(4, add=3)
println("result2 = $result2")
# mysqrt_kwd(4, 3) is not valid
```

Output of Script 1.49: **Adv-Functions-MultArg.jl**

```
result1 = 5.0
result2 = 5.0
```

1.8.4. Computational Speed

Chances are good that you'll be using *Julia* because of what you've heard about its good performance. In this subsection we want to give you a short introduction in measuring the runtime of a function call. We also include a performance comparison of a simple **for** loop in *Julia, R* and *Python* to give you an impression of what *Julia* is capable of.

In Script 1.50 (`Adv-Performance.jl`) we define a simple function **simMean** computing multiple means of randomly generated numbers. The **@timed** command is called a macro and for your purposes we can treat it similar to a function call. It measures the execution time of the subsequent function call in seconds on your system and allows to find potential bottlenecks in your code.

Script 1.50: `Adv-Performance.jl`

```
using Random, Distributions
# set the random seed:
Random.seed!(12345)

function simMean(n, reps)
    ybar = zeros(reps)
    for j in 1:reps
        # sample from normal distribution of size n
        sample = rand(Normal(), n)
        ybar[j] = mean(sample)
    end
    return ybar
end

# call the function and measure time:
n = 100
reps = 10000
stats = @timed simMean(n, reps);
runTime = stats.time
println("runTime = $runTime")
```

Output of Script 1.50: `Adv-Performance.jl`

```
runTime = 0.010652042
```

To get a more reliable estimate of the execution time of a function, it is evaluated multiple times. We use the package **`BenchmarkTools`** for this reason.[33] Figure 1.17 shows the mean runtime of **`simMean`** for different amounts of repetitions and compares it to basic **`for`** loops in *R* and *Python*.[34] This comparison is not completely fair, because you can do many things to improve the performance of *R* and *Python* code. However it shows that *Julia* can give impressive results without having lots of experience in optimizing code.

1.8.5. Outlook

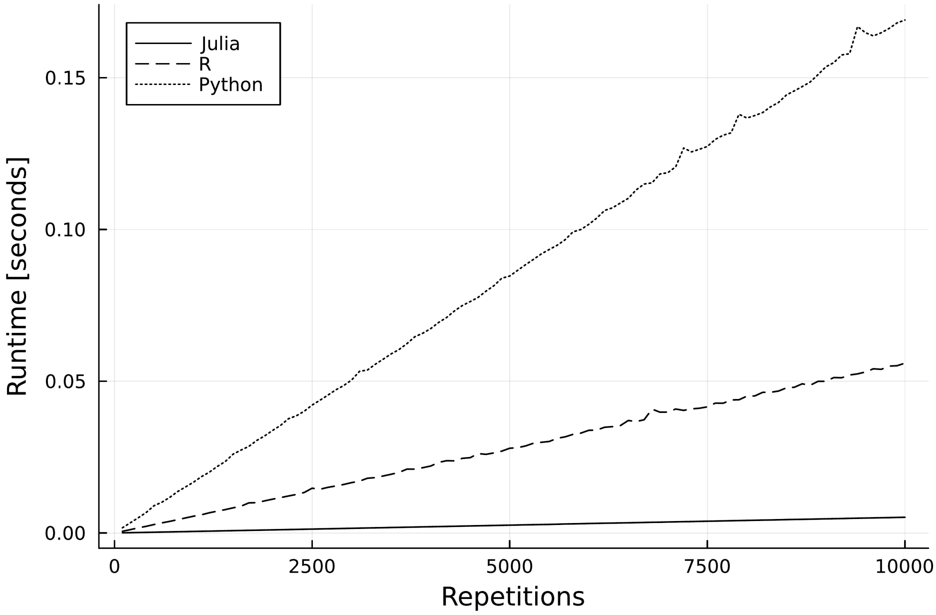
While this section is called "Advanced *Julia*", we have admittedly only scratched the surface of semi-advanced topics. One topic we defer to Chapter 19 is how *Julia* can automatically create formatted reports and publication-ready documents. An example of seriously advanced topics for the real *Julia* geek is to use parallel computing to speed up computations. Since real *Julia* geeks are not the target audience of this book, we will stop to even mention more intimidating possibilities and focus on implementing the most important econometric methods in the most straightforward and pragmatic way.

1.9. Monte Carlo Simulation

Appendix C.2 of Wooldridge (2019) contains a brief introduction to estimators and their properties.[35] In real-world applications, we typically have a data set corresponding to a random sample from a well-defined population. We don't know the population parameters and use the sample to estimate them.

[33] For more information about the package, see Revels (2015).
[34] The code is included in Script 1.51 (`Adv-Performance-Jl-Figure.jl`) in the Appendix.
[35] The stripped-down textbook for Europe and Africa Wooldridge (2014) does not include this either.

Figure 1.17. Computation Time of **simMean**



When we generate a sample using a computer program as we have introduced in Section 1.6.4, we know the population parameters since we had to choose them when making the random draws. We could apply the same estimators to this artificial sample to estimate the population parameters. The tasks would be: (1) Select a population distribution and its parameters. (2) Generate a sample from this distribution. (3) Use the sample to estimate the population parameters.

If this sounds a little insane to you: Don't worry, that would be a healthy first reaction. We obtain a noisy estimate of something we know precisely. But this sort of analysis does in fact make sense. Because we estimate something we actually know, we are able to study the behavior of our estimator very well.

In this book, we mainly use this approach for illustrative and didactic reasons. In state-of-the-art research, it is widely used since it often provides the only way to learn about important features of estimators and statistical tests. A name frequently given to these sorts of analyses is Monte Carlo simulation in reference to the "gambling" involved in generating random samples.

1.9.1. Finite Sample Properties of Estimators

Let's look at a simple example and simulate a situation in which we want to estimate the mean μ of a normally distributed random variable

$$Y \sim \text{Normal}(\mu, \sigma^2) \tag{1.6}$$

using a sample of a given size n. The obvious estimator for the population mean would be the sample average $\bar{Y}$. But what properties does this estimator have? The informed reader immediately knows that the sampling distribution of $\bar{Y}$ is

$$\bar{Y} \sim \text{Normal}\left(\mu, \frac{\sigma^2}{n}\right) \tag{1.7}$$

Simulation provides a way to verify this claim.

Script 1.52 (`Simulate-Estimate.jl`) shows a simulation experiment in action: We set the seed to ensure reproducibility and draw a sample of size $n = 100$ from the population distribution (with the population parameters $\mu = 10$ and $\sigma = 2$).[36] Then, we calculate the sample average as an estimate of μ. We see results for three different samples.

Script 1.52: `Simulate-Estimate.jl`

```
using Distributions, Random

# set the random seed:
Random.seed!(12345)

# set sample size:
n = 100

# draw a sample given the population parameters:
sample1 = rand(Normal(10, 2), n)

# estimate the population mean with the sample average:
estimate1 = mean(sample1)
println("estimate1 = $estimate1\n")

# draw a different sample and estimate again:
sample2 = rand(Normal(10, 2), n)
estimate2 = mean(sample2)
println("estimate2 = $estimate2\n")

# draw a third sample and estimate again:
sample3 = rand(Normal(10, 2), n)
estimate3 = mean(sample3)
print("estimate3: $estimate3")
```

Output of Script 1.52: `Simulate-Estimate.jl`

```
estimate1 = 10.255315396762523

estimate2 = 9.686886737328141

estimate3: 9.958420923636215
```

All sample means $\bar{Y}$ are around the true mean $\mu = 10$ which is consistent with our presumption formulated in Equation 1.7. It is also not surprising that we don't get the exact population parameter – that's the nature of the sampling noise. According to Equation 1.7, the results are expected to have a variance of $\frac{\sigma^2}{n} = 0.04$. Three samples of this kind are insufficient to draw strong conclusions regarding the validity of Equation 1.7. Good Monte Carlo simulation studies should use as many samples as possible.

The code shown in Script 1.53 (`Simulation-Repeated.jl`) uses a **`for`** loop to draw 10,000 samples of size $n = 100$ and calculates the sample average for all of them. After setting the random seed, the empty array **`ybar`** of size 10,000 is initialized using the **`zeros`** command. We will replace these empty array values with the estimates one after another in the loop. In each of these replications $j = 1, 2, \ldots, 10000$, a sample is drawn, its average calculated and stored in position number **`j`** of **`ybar`**. In this way, we end up with a list of 10,000 estimates from different samples. The Script `Simulation-Repeated.jl` does not generate any output.

[36] See Section 1.6.4 for the basics of random number generation.

Script 1.53: `Simulation-Repeated.jl`

```
using Distributions, Random

# set the random seed:
Random.seed!(12345)

# set sample size:
n = 100

# initialize ybar to an array of length r=10000 to later store results:
r = 10000
ybar = zeros(r)

# repeat r times:
for j in 1:r
    # draw a sample and store the sample mean in pos. j=1,... of ybar:
    sample = rand(Normal(10, 2), n)
    ybar[j] = mean(sample)
end
```

Script 1.54 (`Simulation-Repeated-Results.jl`) analyses these 10,000 estimates. Here, we just discuss the output, but you find the complete code in the appendix. The average of **`ybar`** is very close to the presumption $\mu = 10$ from Equation 1.7. Also the simulated sampling variance is close to the theoretical result $\frac{\sigma^2}{n} = 0.04$. Note that the degrees of freedom are adjusted in the function **`var`** to compute the unbiased estimate of the variance. Finally, the estimated density (using a kernel density estimate from the package **`KernelDensity`**) is compared to the theoretical normal distribution. The result is shown in Figure 1.18. The two lines are almost indistinguishable except for the area close to the mode (where the kernel density estimator is known to have problems).

Output of Script 1.54: `Simulation-Repeated-Results.jl`

```
ybar_preview:
[10.2553, 9.6869, 9.9584, 9.9454, 9.9426, 9.7717, 9.7265, 9.9904]

mean_ybar = 10.001370606455168

var_ybar = 0.04090275420375032
```

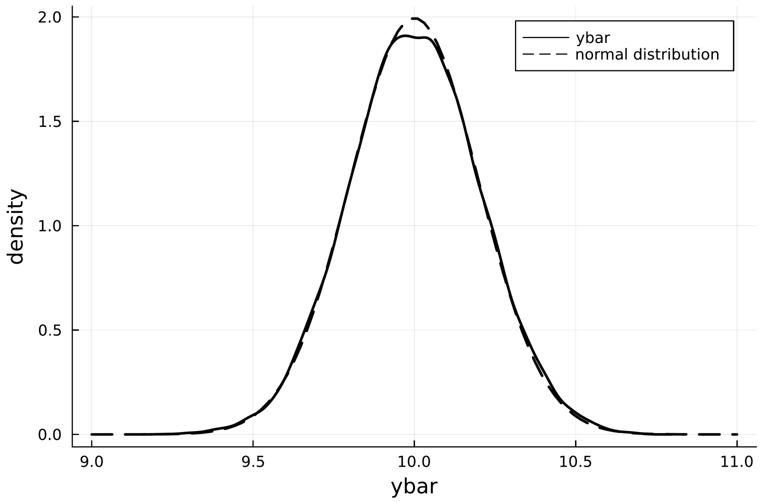
To summarize, the simulation results confirm the theoretical results in Equation 1.7. Mean, variance and density are very close and it seems likely that the remaining tiny differences are due to the fact that we "only" used 10,000 samples.

Remember: for most advanced estimators, such simulations are the only way to study some of their features since it is impossible to derive theoretical results of interest. For us, the simple example hopefully clarified the approach of Monte Carlo simulations and the meaning of the sampling distribution and prepared us for other interesting simulation exercises.

1.9.2. Asymptotic Properties of Estimators

Asymptotic analyses are concerned with large samples and with the behavior of estimators and other statistics as the sample size n increases without bound. For a discussion of these topics, see Wooldridge (2019, Appendix C.3). According to the **law of large numbers**, the sample average $\bar{Y}$ in the above example converges in probability to the population mean μ as $n \to \infty$. In (infinitely) large samples, this implies that $\mathrm{E}(\bar{Y}) \to \mu$ and $\mathrm{Var}(\bar{Y}) \to 0$.

Figure 1.18. Simulated and Theoretical Density of $\bar{Y}$



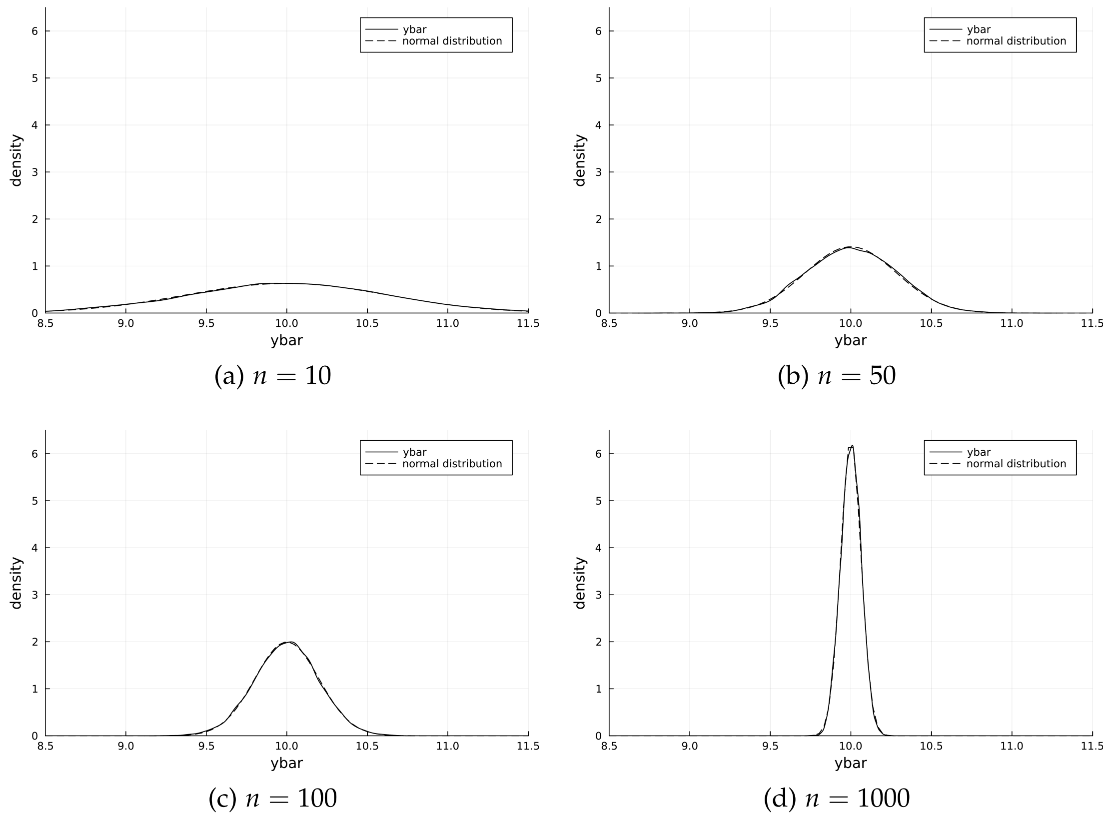
With Monte Carlo simulation, we have a tool to see how this works out in our example. We just have to change the sample size in the code line **`n = 100`** in Script 1.53 (`Simulation-Repeated.jl`) to a different number and rerun the simulation code. Results for $n = 10, 50, 100$, and 1000 are presented in Figure 1.19. Apparently, the variance of $\bar{Y}$ does in fact decrease. The graph of the density for $n = 1000$ is already very narrow and high indicating a small variance. Of course, we cannot actually increase n to infinity without crashing our computer, but it appears plausible that the density will eventually collapse into one vertical line corresponding to $\text{Var}(\bar{Y}) \to 0$ as $n \to \infty$.

In our example for the simulations, the random variable Y was normally distributed, therefore the sample average $\bar{Y}$ was also normal for any sample size. This can also be confirmed in Figure 1.19 where the respective normal densities were added to the graphs as dashed lines. The **central limit theorem** (CLT) claims that as $n \to \infty$, the sample mean $\bar{Y}$ of a random sample will eventually *always* be normally distributed, no matter what the distribution of Y is (unless it is very weird with an infinite variance). This is called convergence in distribution.

Let's check this with a very non-normal distribution, the χ^2 distribution with one degree of freedom. Its density is depicted in Figure 1.20.[37] It looks very different from our familiar bell-shaped normal density. The only line we have to change in the simulation code in Script 1.53 (`Simulation-Repeated.jl`) is **`sample = rand(Normal(10, 2), n)`** which we have to replace with **`sample = rand(Chisq(1), n)`** according to Table 1.6. Figure 1.21 shows the simulated densities for different sample sizes and compares them to the normal distribution with the same mean $\mu = 1$ and standard deviation $\frac{s}{\sqrt{n}} = \sqrt{\frac{2}{n}}$. Note that the scales of the axes now differ between the sub-figures in order to provide a better impression of the shape of the densities. The effect of a decreasing variance works here in exactly the same way as with the normal population. Not surprisingly, the distribution of $\bar{Y}$ is very different from a normal one in small samples like $n = 2$. With increasing sample size, the CLT works its magic and the distribution gets closer to the

[37] A motivated reader will already have figured out that this graph was generated by **`pdf.(Chisq(df), x)`** from the **`Distributions`** package.

Figure 1.19. Density of $\bar{Y}$ with Different Sample Sizes



(a) $n = 10$ (b) $n = 50$

(c) $n = 100$ (d) $n = 1000$

normal bell-shape. For $n = 10000$, the densities hardly differ at all so it's easy to imagine that they will eventually be the same as $n \to \infty$.

1.9.3. Simulation of Confidence Intervals and t Tests

In addition to repeatedly estimating population parameters, we can also calculate confidence intervals and conduct tests on the simulated samples. Here, we present a somewhat advanced simulation routine. The payoff of going through this material is that it might substantially improve our understanding of the workings of statistical inference.

We start from the same example as in Section 1.9.1: In the population, $Y \sim \text{Normal}(10, 4)$. We draw 10,000 samples of size $n = 100$ from this population. For each of the samples we calculate

- The 95% confidence interval and store the limits in **`CIlower`** and **`CIupper`**.
- The p value for the two-sided test of the correct null hypothesis $H_0 : \mu = 10 \Rightarrow$ array **`pvalue1`**
- The p value for the two-sided test of the *incorrect* null hypothesis $H_0 : \mu = 9.5 \Rightarrow$ array **`pvalue2`**

Finally, we calculate the vectors **`reject1`** and **`reject2`** with logical items that are **`true`** if we reject the respective null hypothesis at $\alpha = 5\%$, i.e. if **`pvalue1`** or **`pvalue2`** are smaller than 0.05, respectively. Script 1.56 (`Simulation-Inference.jl`) shows the *Julia* code for these simulations and frequency tables for the results **`reject1`** and **`reject2`**.

Figure 1.20. Density of the χ^2 Distribution with 1 d.f.

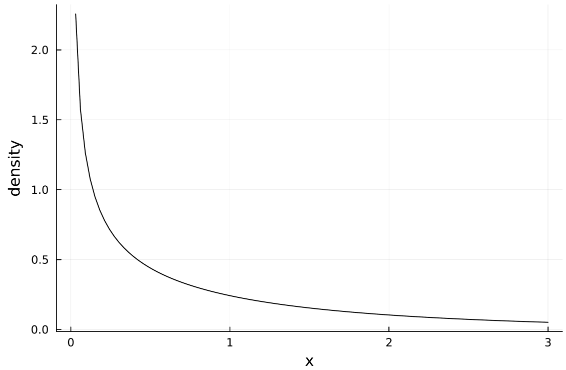
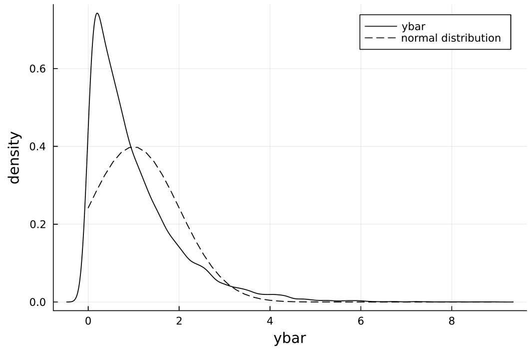
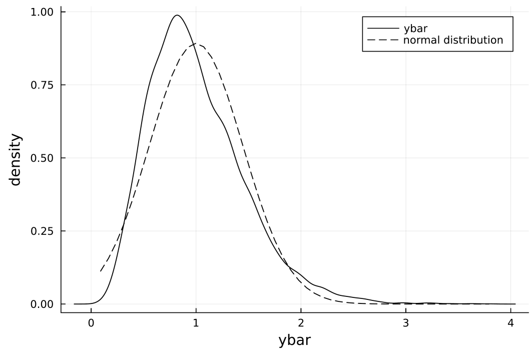


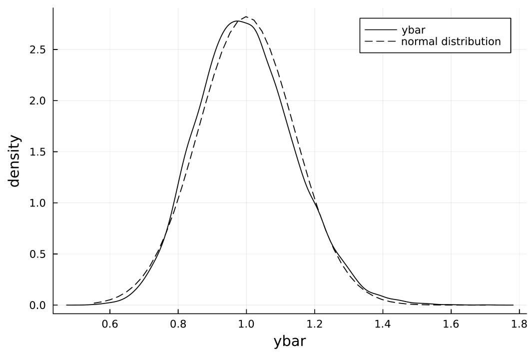
Figure 1.21. Density of $\bar{Y}$ with Different Sample Sizes: χ^2 Distribution



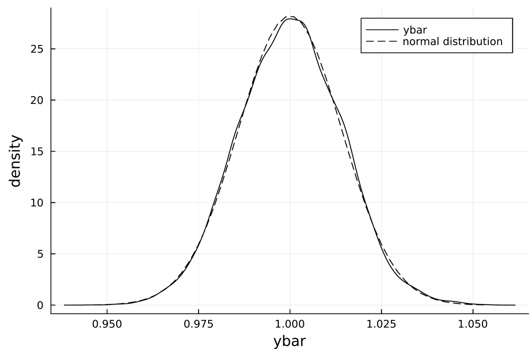
(a) $n = 2$



(b) $n = 10$



(c) $n = 100$



(d) $n = 10000$

If theory and the implementation in *Julia* are accurate, the probability to reject a correct null hypothesis (i.e. to make a Type I error) should be equal to the chosen significance level α. In our simulation, we reject the correct hypothesis in 536 of the 10,000 samples, which amounts to 5.36%.

The probability to reject a false hypothesis is called the power of a test. It depends on many things like the sample size and "how bad" the error of H_0 is, i.e. how far away μ_0 is from the true μ. Theory just tells us that the power is larger than α. In our simulation, the wrong null $H_0 : \mu = 9.5$ is rejected in 69.6% of the samples. The reader is strongly encouraged to tinker with the simulation code to verify the theoretical results that this power increases if μ_0 moves away from 10 and if the sample size n increases.

Figure 1.22 graphically presents the 95% CI for the first 100 simulated samples.[38] Each horizontal line represents one CI. In these first 100 samples, the true null was rejected in five cases. This fact means that for those five samples the CI does not cover $\mu_0 = 10$, see Wooldridge (2019, Appendix C.6) on the relationship between CI and tests. These five cases are drawn in black in the left part of the figure, whereas the others are gray.

The t-test rejects the false null hypothesis $H_0 : \mu = 9.5$ in 71 of the first 100 samples. Their CIs do not cover 9.5 and are drawn in black in the right part of Figure 1.22.

Script 1.56: `Simulation-Inference.jl`

```
using Distributions, Random, HypothesisTests

# set the random seed:
Random.seed!(12345)

# set sample size and MC simulations:
r = 10000
n = 100

# initialize arrays to later store results:
CIlower = zeros(r)
CIupper = zeros(r)
pvalue1 = zeros(r)
pvalue2 = zeros(r)


# repeat r times:
for j in 1:r
    # draw a sample
    sample = rand(Normal(10, 2), n)
    sample_mean = mean(sample)
    sample_sd = std(sample)

    # test the (correct) null hypothesis mu=10:
    testres1 = OneSampleTTest(sample, 10)
    pvalue1[j] = pvalue(testres1)
    cv = quantile(TDist(n - 1), 0.975)
    CIlower[j] = sample_mean - cv * sample_sd / sqrt(n)
    CIupper[j] = sample_mean + cv * sample_sd / sqrt(n)

    # test the (incorrect) null hypothesis mu=9.5 & store the p value:
    testres2 = OneSampleTTest(sample, 9.5)
    pvalue2[j] = pvalue(testres2)

end
```

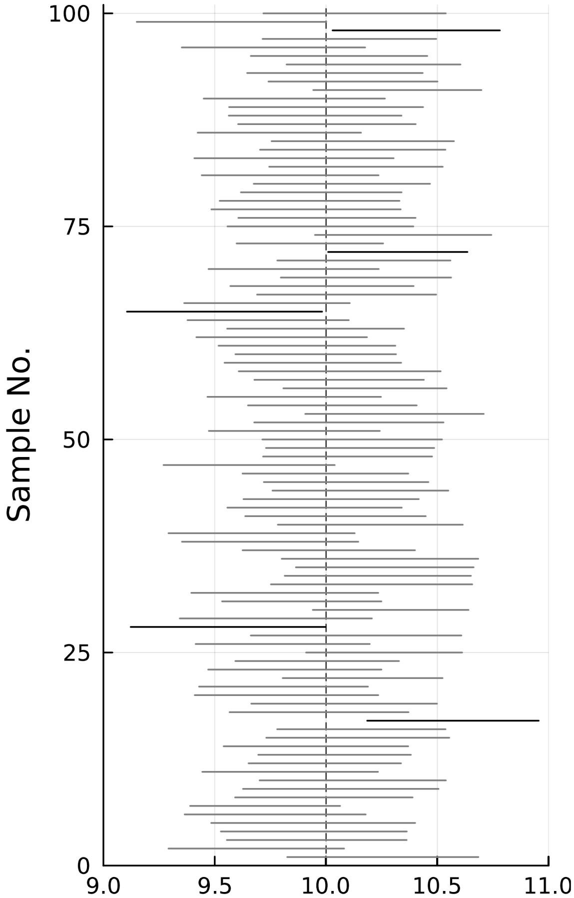
[38] For the sake of completeness, the code for generating these graphs is shown in Appendix IV, Script 1.55 (`Simulation-Inference-Figure.jl`), but most readers will probably not find it important to look at it at this point.

```
# test results as logical value:
reject1 = pvalue1 .<= 0.05
count1_true = count(reject1) # counts true
count1_false = r - count1_true
println("count1_true: $count1_true\n")
println("count1_false: $count1_false\n")

reject2 = pvalue2 .<= 0.05
count2_true = count(reject2)
count2_false = r - count2_true
println("count2_true: $count2_true\n")
println("count2_false: $count2_false")
```

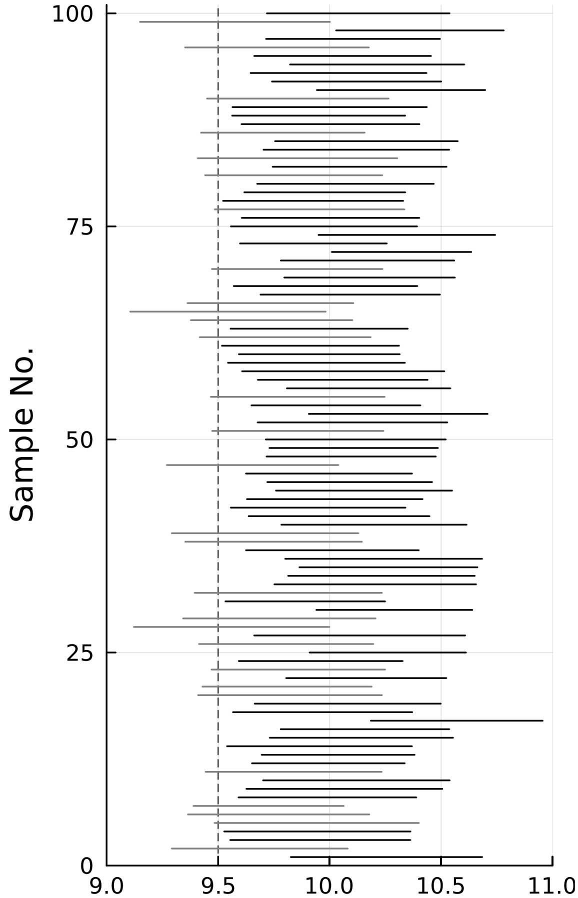
Output of Script 1.56: `Simulation-Inference.jl`

```
count1_true: 536

count1_false: 9464

count2_true: 6962

count2_false: 3038
```

Figure 1.22. Simulation Results: First 100 Confidence Intervals



(a) correct H_0



(b) incorrect H_0

Part I.

Regression Analysis with Cross-Sectional Data

2. The Simple Regression Model

2.1. Simple OLS Regression

We are concerned with estimating the population parameters β_0 and β_1 of the simple linear regression model

$$y = \beta_0 + \beta_1 x + u \tag{2.1}$$

from a random sample of y and x. According to Wooldridge (2019, Section 2.2), the ordinary least squares (OLS) estimators are

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x} \tag{2.2}$$

$$\hat{\beta}_1 = \frac{\mathrm{Cov}(x,y)}{\mathrm{Var}(x)}. \tag{2.3}$$

Based on these estimated parameters, the OLS regression line is

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x. \tag{2.4}$$

For a given sample, we just need to calculate the four statistics $\bar{y}$, $\bar{x}$, $\mathrm{Cov}(x,y)$, and $\mathrm{Var}(x)$ and plug them into these equations. We already know how to make these calculations in *Julia*, see Section 1.5. Let's do it!

Wooldridge, Example 2.3: CEO Salary and Return on Equity

We are using the data set `CEOSAL1` we already analyzed in Section 1.5. We consider the simple regression model

$$\texttt{salary} = \beta_0 + \beta_1 \texttt{roe} + u$$

where `salary` is the salary of a CEO in thousand dollars and `roe` is the return on investment in percent. In Script 2.1 (`Example-2-3.jl`), we first load the packages and the data set. We also calculate the four statistics we need for Equations 2.2 and 2.3 so we can reproduce the OLS formulas by hand. Finally, the parameter estimates are calculated.
So the OLS regression line is

$$\widehat{\texttt{salary}} = 963.1913 + 18.50119 \cdot \texttt{roe}.$$

Script 2.1: `Example-2-3.jl`

```
using WooldridgeDatasets, DataFrames, Statistics

ceosal1 = DataFrame(wooldridge("ceosal1"))
x = ceosal1.roe
y = ceosal1.salary

# ingredients to the OLS formulas:
cov_xy = cov(x, y)
var_x = var(x)
x_bar = mean(x)
y_bar = mean(y)

# manual calculation of OLS coefficients:
b1 = cov_xy / var_x
b0 = y_bar - b1 * x_bar
println("b1 = $b1\n")
println("b0 = $b0")
```

Output of Script 2.1: `Example-2-3.jl`

```
b1 = 18.50118634521492

b0 = 963.191336472558
```

While calculating OLS coefficients using this pedestrian approach is straightforward, there is a more convenient way to do it. Given the importance of OLS regression, it is not surprising that many *Julia* packages have a specialized command to do the calculations automatically. In the following chapters, we will often use the package **`GLM`** to apply linear regression and other econometric methods.[1] When working with **`GLM`**, the first line of code often is:

```
using GLM
```

If the data frame **`sample`** contains the values of the dependent variable in column **`y`** and those of the regressor in the column **`x`**, we can calculate the OLS coefficients as

```
reg = lm(@formula(y ~ x), sample)
```

The first argument including **`y ~ x`** is called a **`formula`**. Essentially, it means that we want to model a left-hand side variable **`y`** to be explained by a right-hand side variable **`x`** in a linear fashion. We will discuss more general model formulae in Section 6.1. The second argument **`sample`** refers to the data that is used for the calculation of OLS coefficients.

Finally, all kind of results are assigned to the variable **`reg`**. The name could of course be anything, for example **`yummy_chocolate_chip_cookies`**, but choosing telling variable names makes our life easier. The referenced object does not only include the OLS coefficients, but also information on standard errors and much more we will get to know and use later on.

[1]For more information about the package, see Bates and other contributors (2012) or `https://juliastats.org/GLM.jl/stable/`.

Wooldridge, Example 2.3: CEO Salary and Return on Equity (*cont'ed*)

In Script 2.2 (`Example-2-3-2.jl`), we repeat the analysis we have already done manually. Besides the import of the data, there are only a few lines of code. The output shows how to access both estimated parameters with **`coef(reg)`**: $\hat{\beta}_0$ is the first, and $\hat{\beta}_1$ is second element in the vector **`b`**. The values are the same we already calculated except for different rounding in the output.

Script 2.2: `Example-2-3-2.jl`

```
using WooldridgeDatasets, DataFrames, GLM

ceosal1 = DataFrame(wooldridge("ceosal1"))

reg = lm(@formula(salary ~ roe), ceosal1)
b = coef(reg)
println("b = $b")
```

Output of Script 2.2: `Example-2-3-2.jl`

```
b = [963.191336472557, 18.50118634521497]
```

From now on, we will rely on the built-in routine in **`GLM`** instead of doing the calculations manually. It is not only more convenient for calculating the coefficients, but also for further analyses as we will see soon.

Given the results from a regression, plotting the regression line is straightforward. In this case, we simply supply the regressor **`roe`** and the predicted values (available under **`predict(reg)`**) and connect them by a line with **`plot`**. We also use the command **`scatter`** to add points to the graph.

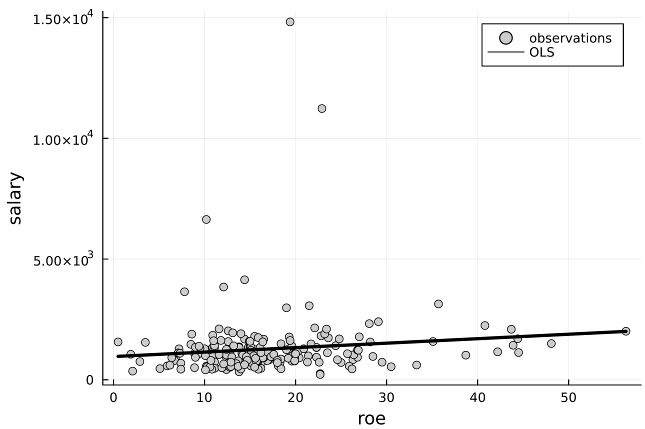
Wooldridge, Example 2.3: CEO Salary and Return on Equity (*cont'ed*)

Script 2.3 (`Example-2-3-3.jl`) demonstrates how to store the regression results in a variable **`reg`** and then use the resulting fitted values as an argument to **`plot`** to add the regression line to the scatter plot. It generates Figure 2.1.

Script 2.3: `Example-2-3-3.jl`

```
using WooldridgeDatasets, DataFrames, GLM, Plots

ceosal1 = DataFrame(wooldridge("ceosal1"))

reg = lm(@formula(salary ~ roe), ceosal1)

# scatter plot and fitted values:
fitted_values = predict(reg)
scatter(ceosal1.roe, ceosal1.salary, color=:grey80, label="observations")
plot!(ceosal1.roe, fitted_values, color=:black, linewidth=3, label="OLS")
xlabel!("roe")
ylabel!("salary")
savefig("JlGraphs/Example-2-3-3.pdf")

# instead of scatter, you can also use:
# plot(ceosal1.roe, ceosal1.salary, label="observations", seriestype=:scatter)
```

Figure 2.1. OLS Regression Line for Example 2-3



Wooldridge, Example 2.4: Wage and Education

We are using the data set `WAGE1`. We are interested in studying the relation between education and wage, and our regression model is

$$\texttt{wage} = \beta_0 + \beta_1 \texttt{education} + u.$$

In Script 2.4 (`Example-2-4.jl`), we analyze the data and find that the OLS regression line is

$$\widehat{\texttt{wage}} = -0.90 + 0.54 \cdot \texttt{education}.$$

One additional year of education is associated with an increase of the typical wage by about 54 cents an hour.

Script 2.4: Example-2-4.jl
```
using WooldridgeDatasets, DataFrames, GLM

wage1 = DataFrame(wooldridge("wage1"))

reg = lm(@formula(wage ~ educ), wage1)
b = coef(reg)
println("b = $b")
```

Output of Script 2.4: Example-2-4.jl
```
b = [-0.9048516119571958, 0.5413592546651733]
```

Wooldridge, Example 2.5: Voting Outcomes and Campaign Expenditures

The data set `VOTE1` contains information on campaign expenditures (`shareA` = share of campaign spending in %) and election outcomes (`voteA` = share of vote in %). The regression model

$$\texttt{voteA} = \beta_0 + \beta_1 \texttt{shareA} + u$$

is estimated in Script 2.5 (`Example-2-5.jl`). The OLS regression line turns out to be

$$\widehat{\texttt{voteA}} = 26.81 + 0.464 \cdot \texttt{shareA}.$$

The scatter plot with the regression line generated in the code is shown in Figure 2.2.

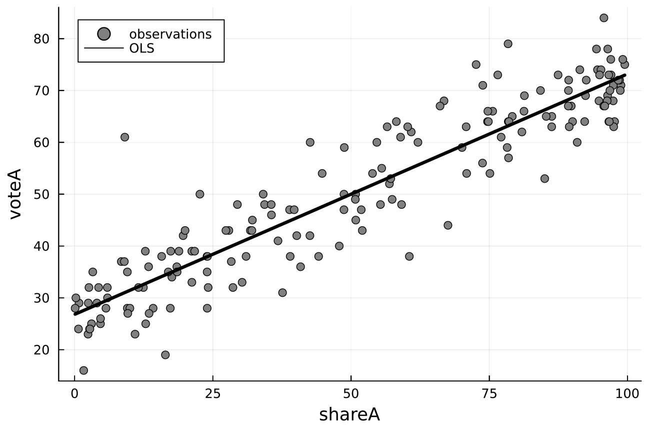
Script 2.5: Example-2-5.jl
```
using WooldridgeDatasets, DataFrames, GLM, Plots

vote1 = DataFrame(wooldridge("vote1"))

# OLS regression:
reg = lm(@formula(voteA ~ shareA), vote1)
b = coef(reg)
println("b = $b")

# scatter plot and fitted values:
fitted_values = predict(reg)
scatter(vote1.shareA, vote1.voteA,
    color=:grey, label="observations", legend=:topleft)
plot!(vote1.shareA, fitted_values, color=:black, linewidth=3, label="OLS")
xlabel!("shareA")
ylabel!("voteA")
savefig("JlGraphs/Example-2-5.pdf")
```

Output of Script 2.5: Example-2-5.jl
```
b = [26.812214128680353, 0.4638269122908861]
```

2.2. Coefficients, Fitted Values, and Residuals

The object returned by the function **`lm`** contains other important information on the regression and is the basis for further calculations. After defining the regression results object **`reg`** in Script 2.2 (`Example-2-3-2.jl`), we can access the OLS coefficients with

Figure 2.2. OLS Regression Line for Example 2-5




```
coef(reg)
```

$\hat{\beta}_0$ is the first, and $\hat{\beta}_1$ is second element in the returned vector, so you can access the parameters separately by using the position. For example, in Script 2.2 (`Example-2-3-2.jl`) you can access intercept and slope parameter by

```
b[1] # intercept
b[2] # slope parameter
```

To attach names to calculated coefficients, you can also use **coeftable** instead of **coef**. See 2.6 (`Example-2-6.jl`) for an example.

Given these parameter estimates, calculating the predicted values $\hat{y}_i$ and residuals $\hat{u}_i$ for each observation $i = 1, ..., n$ is easy:

$$\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 \cdot x_i \tag{2.5}$$

$$\hat{u}_i = y_i - \hat{y}_i \tag{2.6}$$

If the values of the dependent and independent variables are stored in a data frame **sample** as **y** and **x**, respectively, we can estimate the model and do the calculations of these equations for all observations jointly using the code

```
reg = lm(@formula(y ~ x), sample)
b = coef(reg)
y_hat = b[1] .+ b[2] .* sample.x
u_hat = sample.y .- y_hat
```

We can also use a more black box approach which will give exactly the same results using **predict** and **residuals** on the regression results object:

```
reg = lm(@formula(y ~ x), sample)
y_hat = predict(reg)
u_hat = residuals(reg)
```

Wooldridge, Example 2.6: CEO Salary and Return on Equity

We extend the regression example on the return on equity of a firm and the salary of its CEO in Script 2.6 (`Example-2-6.jl`). After the OLS regression, we calculate fitted values and residuals. A table similar to Wooldridge (2019, Table 2.2) is generated displaying the values for the first 10 observations.

Script 2.6: `Example-2-6.jl`

```
using WooldridgeDatasets, DataFrames, GLM

ceosal1 = DataFrame(wooldridge("ceosal1"))

# OLS regression:
reg = lm(@formula(salary ~ roe), ceosal1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg\n")

# obtain predicted values and residuals:
salary_hat = predict(reg)
u_hat = residuals(reg)

# Wooldridge, Table 2.2:
table = DataFrame(roe=ceosal1.roe,
    salary=ceosal1.salary,
    salary_hat=salary_hat,
    u_hat=u_hat)
table_preview = first(table, 10)
println("table_preview: \n$table_preview")
```

Output of Script 2.6: `Example-2-6.jl`

```
table_reg:

                Coef.  Std. Error     t  Pr(>|t|)  Lower 95%  Upper 95%

(Intercept)  963.191     213.24    4.52    <1e-04   542.79    1383.59
roe           18.5012     11.1233  1.66    0.0978    -3.4282    40.4306


table_preview:
10×4 DataFrame
 Row │ roe      salary  salary_hat  u_hat
     │ Float64  Int64   Float64     Float64
─────┼─────────────────────────────────────────────
   1 │    14.1    1095     1224.06  -129.058
   2 │    10.9    1001     1164.85  -163.854
   3 │    23.5    1122     1397.97  -275.969
   4 │     5.9     578     1072.35  -494.348
   5 │    13.8    1368     1218.51   149.492
   6 │    20.0    1145     1333.22  -188.215
   7 │    16.4    1078     1266.61  -188.611
   8 │    16.3    1094     1264.76  -170.761
   9 │    10.5    1237     1157.45    79.5462
  10 │    26.3     833     1449.77  -616.773
```

Wooldridge (2019, Section 2.3) presents and discusses three properties of OLS statistics which we will confirm for an example.

$$\sum_{i=1}^{n} \hat{u}_i = 0 \quad \Rightarrow \quad \bar{\hat{u}}_i = 0 \tag{2.7}$$

$$\sum_{i=1}^{n} x_i \hat{u}_i = 0 \quad \Rightarrow \quad \text{Cov}(x_i, \hat{u}_i) = 0 \tag{2.8}$$

$$\bar{y} = \hat{\beta}_0 + \hat{\beta}_1 \cdot \bar{x} \tag{2.9}$$

Wooldridge, Example 2.7: Wage and Education

We already know the regression results when we regress wage on education from Example 2.4. In Script 2.7 (`Example-2-7.jl`), we calculate fitted values and residuals to confirm the three properties from Equations 2.7 through 2.9. Note that *Julia* does all calculations in "double precision" implying that it is accurate for at least 15 significant digits. The output that checks the first property shows that the average residual is `5.943703493050267e-16` which in scientific notation means $5.943703493050267 \cdot 10^{-16} = 0.0000000000000005943703493050267$. The reason it is not exactly equal to 0 is a rounding error in the 17th digit. The same holds for the second property: The covariance between the regressor and the residual is zero except for minimal rounding error. Note that running Script 2.7 (`Example-2-7.jl`) will give you the same accurate digits, but the digits with rounding error will differ. The third property is also confirmed: If we plug the average value of the regressor into the regression line formula, we get the average value of the dependent variable.

Script 2.7: `Example-2-7.jl`

```
using WooldridgeDatasets, DataFrames, GLM, Statistics

wage1 = DataFrame(wooldridge("wage1"))
reg = lm(@formula(wage ~ educ), wage1)

# obtain coefficients, predicted values and residuals:
b = coef(reg)
wage_hat = predict(reg)
u_hat = residuals(reg)

# confirm property (1):
u_hat_mean = mean(u_hat)
println("u_hat_mean = $u_hat_mean\n")

# confirm property (2):
educ_u_cov = cov(wage1.educ, u_hat)
println("educ_u_cov = $educ_u_cov\n")

# confirm property (3):
educ_mean = mean(wage1.educ)
wage_pred = b[1] + b[2] * educ_mean
println("wage_pred = $wage_pred\n")

wage_mean = mean(wage1.wage)
println("wage_mean = $wage_mean")
```

Output of Script 2.7: `Example-2-7.jl`

```
u_hat_mean = 5.943703493050267e-16

educ_u_cov = 9.828178542739973e-15

wage_pred = 5.896102674787035

wage_mean = 5.896102674787035
```

2.3. Goodness of Fit

The total sum of squares (SST), explained sum of squares (SSE) and residual sum of squares (SSR) can be written as

$$\text{SST} = \sum_{i=1}^{n}(y_i - \overline{y})^2 = (n-1)\cdot \text{Var}(y) \tag{2.10}$$

$$\text{SSE} = \sum_{i=1}^{n}(\hat{y}_i - \overline{y})^2 = (n-1)\cdot \text{Var}(\hat{y}) \tag{2.11}$$

$$\text{SSR} = \sum_{i=1}^{n}(\hat{u}_i - 0)^2 = (n-1)\cdot \text{Var}(\hat{u}) \tag{2.12}$$

where $\text{Var}(x)$ is the sample variance $\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2$.

Wooldridge (2019, Equation 2.38) defines the coefficient of determination in terms of these terms. Because $(n-1)$ cancels out, it can be equivalently written as

$$R^2 = \frac{\text{Var}(\hat{y})}{\text{Var}(y)} = 1 - \frac{\text{Var}(\hat{u})}{\text{Var}(y)}. \tag{2.13}$$

Wooldridge, Example 2.8: CEO Salary and Return on Equity

In the regression already studied in Example 2.6, the coefficient of determination is 0.0132. This is calculated in the two ways of Equation 2.13 in Script 2.8 (`Example-2-8.jl`). In addition, it is calculated as the squared correlation coefficient of y and $\hat{y}$. Not surprisingly, all versions of these calculations produce the same result (they are not exactly equal to each other because of the rounding error in the 17$^{\text{th}}$ digit).

Script 2.8: `Example-2-8.jl`

```
using WooldridgeDatasets, DataFrames, GLM, Statistics

ceosal1 = DataFrame(wooldridge("ceosal1"))

# OLS regression:
reg = lm(@formula(salary ~ roe), ceosal1)

# obtain predicted values and residuals:
sal_hat = predict(reg)
u_hat = residuals(reg)

# calculate R^2 in three different ways:
sal = ceosal1.salary
R2_a = var(sal_hat) / var(sal)
R2_b = 1 - var(u_hat) / var(sal)
R2_c = cor(sal, sal_hat)^2

println("R2_a = $R2_a\n")
println("R2_b = $R2_b\n")
println("R2_c = $R2_c")
```

Output of Script 2.8: `Example-2-8.jl`

```
R2_a = 0.013188624081034168

R2_b = 0.013188624081034162

R2_c = 0.013188624081034099
```

Many interesting results for a regression can be computed automatically with the output of **`lm`**, and we show this for the coefficient of determination. As demonstrated in Script 2.9 (`Example-2-9.jl`), the function **`r2`** calculates the coefficient of determination just with the output object of **`lm`** .

When we are interested in the coefficients and their significance, we will often use the function **`coeftable`** for a compact presentation of results. This is demonstrated with the object **`table_reg`** in Script 2.9 (`Example-2-9.jl`). We will discuss the details of this additional information later.

Wooldridge, Example 2.9: Voting Outcomes and Campaign Expenditures

We already know the OLS coefficients to be $\hat{\beta}_0 = 26.8122$ and $\hat{\beta}_1 = 0.4638$ in the voting example (Script 2.5 (`Example-2-5.jl`)). These values are again found in the output of Script 2.9 (`Example-2-9.jl`). The coefficient of determination is reported as **`r2_automatic`** to be $R^2 = 0.856$.

Script 2.9: Example-2-9.jl

```
using WooldridgeDatasets, DataFrames, GLM

vote1 = DataFrame(wooldridge("vote1"))

# OLS regression:
reg = lm(@formula(voteA ~ shareA), vote1)

# print results using coeftable:
table_reg = coeftable(reg)
println("table_reg: \n$table_reg\n")

# accessing R^2:
r2_automatic = r2(reg)
println("r2_automatic = $r2_automatic")
```

Output of Script 2.9: Example-2-9.jl

```
table_reg:

                Coef.   Std. Error      t  Pr(>|t|)  Lower 95%  Upper 95%

(Intercept)  26.8122     0.887215   30.22    <1e-69  25.0609    28.5635
shareA        0.463827   0.0145397  31.90    <1e-73   0.435127   0.492527


r2_automatic = 0.8561408655827665
```

2.4. Nonlinearities

For the estimation of logarithmic or semi-logarithmic models, the respective formula can be directly entered into the specification of `lm(@formula(...))` as demonstrated in Examples 2.10 and 2.11. For the interpretation as percentage effects and elasticities, see Wooldridge (2019, Section 2.4).

Wooldridge, Example 2.10: Wage and Education

Compared to Example 2.7, we simply change the command for the estimation to account for a logarithmic specification as shown in Script 2.10 (`Example-2-10.jl`). The semi-logarithmic specification implies that wages are higher by about 8.3% for individuals with an additional year of education.

Script 2.10: Example-2-10.jl

```
using WooldridgeDatasets, DataFrames, GLM

wage1 = DataFrame(wooldridge("wage1"))

# estimate log-level model:
reg = lm(@formula(log(wage) ~ educ), wage1)
b = coef(reg)
println("b = $b")
```

Output of Script 2.10: Example-2-10.jl

```
b = [0.5837726746718852, 0.08274436586375138]
```

Wooldridge, Example 2.11: CEO Salary and Firm Sales

We study the relationship between the sales of a firm and the salary of its CEO using a log-log specification. The results are shown in Script 2.11 (`Example-2-11.jl`). If the sales increase by 1%, the salary of the CEO tends to increase by 0.257%.

Script 2.11: `Example-2-11.jl`

```
using WooldridgeDatasets, DataFrames, GLM

ceosal1 = DataFrame(wooldridge("ceosal1"))

# estimate log-log model:
reg = lm(@formula(log(salary) ~ log(sales)), ceosal1)
b = coef(reg)
println("b = $b")
```

Output of Script 2.11: `Example-2-11.jl`

```
b = [4.821996478164936, 0.2566716916641498]
```

2.5. Regression through the Origin and Regression on a Constant

Wooldridge (2019, Section 2.6) discusses models without an intercept. This implies that the regression line is forced to go through the origin. In *Julia*, we can suppress the constant which is otherwise implicitly added to a formula by specifying

```
reg = lm(@formula(y ~ 0 + x), sample)
```

instead of `reg = lm(@formula(y ~ x), sample)`. The result is a model which only has a slope parameter.

Another topic discussed in this section is a linear regression model without a slope parameter, i.e. with a constant only. In this case, the estimated constant will be the sample average of the dependent variable. This can be implemented in *Julia* using the code

```
reg = lm(@formula(y ~ 1), sample)
```

Both special kinds of regressions are implemented in Script 2.12 (`SLR-Origin-Const.jl`) for the example of the CEO salary and ROE we already analyzed in Example 2.8 and others. The resulting regression lines are plotted in Figure 2.3 which was generated using the last lines of code shown in Script 2.12 (`SLR-Origin-Const.jl`).

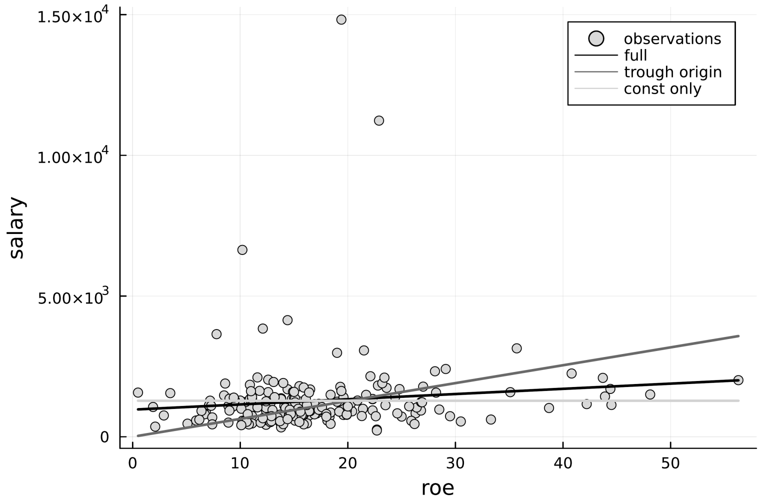
Script 2.12: `SLR-Origin-Const.jl`

```
using WooldridgeDatasets, DataFrames, GLM, Plots, Statistics

ceosal1 = DataFrame(wooldridge("ceosal1"))

# usual OLS regression:
reg1 = lm(@formula(salary ~ roe), ceosal1)
b1 = coef(reg1)
println("b1 = $b1\n")

# regression without intercept (through origin):
reg2 = lm(@formula(salary ~ 0 + roe), ceosal1)
b2 = coef(reg2)
println("b2 = $b2\n")

# regression without slope (on a constant):
reg3 = lm(@formula(salary ~ 1), ceosal1)
b3 = coef(reg3)
println("b3 = $b3\n")

# average y:
sal_mean = mean(ceosal1.salary)
println("sal_mean = $sal_mean")

# scatter plot and fitted values:
scatter(ceosal1.roe, ceosal1.salary, color="grey85", label="observations")
plot!(ceosal1.roe, predict(reg1), linewidth=2,
    color="black", label="full")
plot!(ceosal1.roe, predict(reg2), linewidth=2,
    color="dimgrey", label="trough origin")
plot!(ceosal1.roe, predict(reg3), linewidth=2,
    color="lightgrey", label="const only")
xlabel!("roe")
ylabel!("salary")
savefig("JlGraphs/SLR-Origin-Const.pdf")
```

Output of Script 2.12: `SLR-Origin-Const.jl`

```
b1 = [963.191336472557, 18.50118634521497]

b2 = [63.537955204261635]

b3 = [1281.1196172248804]

sal_mean = 1281.1196172248804
```

Figure 2.3. Regression through the Origin and on a Constant



2.6. Expected Values, Variances, and Standard Errors

Wooldridge (2019) discusses the role of five assumptions under which the OLS parameter estimators have desirable properties. In short form they are

- **SLR.1**: Linear population regression function: $y = \beta_0 + \beta_1 x + u$
- **SLR.2**: Random sampling of x and y from the population
- **SLR.3**: Variation in the sample values $x_1, ..., x_n$
- **SLR.4**: Zero conditional mean: $\mathrm{E}(u|x) = 0$
- **SLR.5**: Homoscedasticity: $\mathrm{Var}(u|x) = \sigma^2$

Based on those, Wooldridge (2019) shows in Section 2.5:

- **Theorem 2.1**: Under **SLR.1 – SLR.4**, OLS parameter estimators are unbiased.
- **Theorem 2.2**: Under **SLR.1 – SLR.5**, OLS parameter estimators have a specific sampling variance.

Because the formulas for the sampling variance involve the variance of the error term, we also have to estimate it using the unbiased estimator

$$\hat{\sigma}^2 = \frac{1}{n-2} \cdot \sum_{i=1}^{n} \hat{u}_i^2 = \frac{n-1}{n-2} \cdot \mathrm{Var}(\hat{u}_i), \tag{2.14}$$

where $Var(\hat{u}_i) = \frac{1}{n-1} \cdot \sum_{i=1}^{n} \hat{u}_i^2$ is the usual sample variance. We have to use the degrees-of-freedom adjustment to account for the fact that we estimated the two parameters $\hat{\beta}_0$ and $\hat{\beta}_1$ for constructing the residuals. Its square root $\hat{\sigma} = \sqrt{\hat{\sigma}^2}$ is called **standard error of the regression (SER)** by Wooldridge (2019).

The **standard errors (SE) of the estimators** are

$$\text{se}(\hat{\beta}_0) = \sqrt{\frac{\hat{\sigma}^2\overline{x^2}}{\sum_{i=1}^{n}(x-\overline{x})^2}} = \frac{1}{\sqrt{n-1}} \cdot \frac{\hat{\sigma}}{\text{sd}(x)} \cdot \sqrt{\overline{x^2}} \tag{2.15}$$

$$\text{se}(\hat{\beta}_1) = \sqrt{\frac{\hat{\sigma}^2}{\sum_{i=1}^{n}(x-\overline{x})^2}} = \frac{1}{\sqrt{n-1}} \cdot \frac{\hat{\sigma}}{\text{sd}(x)} \tag{2.16}$$

where $\text{sd}(x)$ is the sample standard deviation $\sqrt{\frac{1}{n-1} \cdot \sum_{i=1}^{n}(x_i-\overline{x})^2}$.

In *Julia*, we can obviously do the calculations of Equations 2.15 through 2.16 explicitly. But the output of the **`coeftable`** command for linear regression results, which we discovered in Section 2.3, already contains the results. We use the following example to calculate the results in both ways to open the black box of the canned routine and convince ourselves that from now on we can rely on it.

Wooldridge, Example 2.12: Student Math Performance and the School Lunch Program

Using the data set `MEAP93`, we regress a math performance score of schools on the share of students eligible for a federally funded lunch program. Wooldridge (2019) uses this example to demonstrate the importance of assumption SLR.4 and warns us against interpreting the regression results in a causal way. Here, we merely use the example to demonstrate the calculation of standard errors.

Script 2.13 (`Example-2-12.jl`) first calculates the SER manually using the fact that the residuals $\hat{u}$ are available as **`residuals(reg)`**. Then, the SE of the parameters are calculated according to Equations 2.15 and 2.16, where the regressor is addressed as the variable in the data frame **`meap93.lnchprg`**. Finally, we see the output of **`coeftable`**. The SE of the parameters are reported in the second column of the regression table, next to the parameter estimates. We will look at the other columns in Chapter 4. All values are exactly the same as the manual results.

Script 2.13: `Example-2-12.jl`

```
using WooldridgeDatasets, DataFrames, GLM, Statistics

meap93 = DataFrame(wooldridge("meap93"))

# estimate the model and save the results as reg:
reg = lm(@formula(math10 ~ lnchprg), meap93)

# number of obs.:
n = nobs(reg)

# SER:
u_hat_var = var(residuals(reg))
SER = sqrt(u_hat_var) * sqrt((n - 1) / (n - 2))
println("SER = $SER\n")

# SE of b0 and b1, respectively:
lnchprg_sq_mean = mean(meap93.lnchprg .^ 2)
lnchprg_var = var(meap93.lnchprg)
b0_se = SER / (sqrt(lnchprg_var) * sqrt(n - 1)) * sqrt(lnchprg_sq_mean)
b1_se = SER / (sqrt(lnchprg_var) * sqrt(n - 1))
println("b0_se = $b0_se\n")
println("b1_se = $b1_se\n")

# automatic calculations:
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Output of Script 2.13: `Example-2-12.jl`

```
SER = 9.565938459482759

b0_se = 0.9975823856755017

b1_se = 0.03483933425836962

table_reg:

                 Coef.  Std. Error      t  Pr(>|t|)  Lower 95%  Upper 95%

(Intercept)  32.1427     0.997582   32.22    <1e-99  30.1816    34.1038
lnchprg      -0.318864   0.0348393  -9.15    <1e-17  -0.387352  -0.250376
```

2.7. Monte Carlo Simulations

In this section, we use Monte Carlo simulation experiments to revisit many of the topics covered in this chapter. It can be skipped but can help quite a bit to grasp the concepts of estimators, estimates, unbiasedness, the sampling variance of the estimators, and the consequences of violated assumptions. Remember that the concept of Monte Carlo simulations was introduced in Section 1.9.

2.7.1. One Sample

In Section 1.9, we used simulation experiments to analyze the features of a simple mean estimator. We also discussed the sampling from a given distribution, the random seed and simple examples. We can use exactly the same strategy to analyze OLS parameter estimators.

Script 2.14 (SLR-Sim-Sample.jl) shows how to draw a sample which is consistent with Assumptions SLR.1 through SLR.5. We simulate a sample of size $n = 1000$ with population parameters $\beta_0 = 1$ and $\beta_1 = 0.5$. We set the standard deviation of the error term u to $\sigma = 2$. Obviously, these parameters can be freely chosen and every reader is strongly encouraged to play around.

Script 2.14: SLR-Sim-Sample.jl

```
using Random, GLM, DataFrames, Distributions, Statistics, Plots

# set the random seed:
Random.seed!(12345)

# set sample size:
n = 1000

# set true parameters (betas and sd of u):
beta0 = 1
beta1 = 0.5
su = 2

# draw a sample of size n:
x = rand(Normal(4, 1), n)
u = rand(Normal(0, su), n)
y = beta0 .+ beta1 .* x .+ u
df = DataFrame(y=y, x=x)

# estimate parameters by OLS:
reg = lm(@formula(y ~ x), df)
b = coef(reg)
println("b = $b\n")

# features of the sample for the variance formula:
x_sq_mean = mean(x .^ 2)
println("x_sq_mean = $x_sq_mean\n")
x_var = sum((x .- mean(x)) .^ 2)
println("x_var = $x_var")

# graph:
x_range = range(0, 8, length=100)
scatter(x, y, color="lightgrey", ylim=[-2, 10],
    label="sample", alpha=0.7, markerstrokecolor=:white)
plot!(x_range, beta0 .+ beta1 .* x_range, color="black",
    linestyle=:solid, linewidth=2, label="pop. regr. fct.")
plot!(x_range, coef(reg)[1] .+ coef(reg)[2] .* x_range, color="grey",
    linestyle=:solid, linewidth=2, label="OLS regr. fct.")
```

```
xlabel!("x")
ylabel!("y")
savefig("JlGraphs/SLR-Sim-Sample.pdf")
```

Output of Script 2.14: `SLR-Sim-Sample.jl`

```
b = [1.0941139603852943, 0.49071895330338017]

x_sq_mean = 16.855828584405046

x_var = 1069.5006680128563
```

Then a random sample of x and y is drawn in three steps:

- A sample of regressors x is drawn from an arbitrary distribution. The only thing we have to make sure to stay consistent with Assumption SLR.3 is that its variance is strictly positive. We choose a normal distribution with mean 4 and a standard deviation of 1.
- A sample of error terms u is drawn according to Assumptions SLR.4 and SLR.5: It has a mean of zero, and both the mean and the variance are unrelated to x. We simply choose a normal distribution with mean 0 and standard deviation $\sigma = 2$ for all 1000 observations independent of x. In Sections 2.7.3 and 2.7.4 we will adjust this to simulate the effects of a violation of these assumptions.
- Finally, we generate the dependent variable y according to the population regression function specified in Assumption SLR.1.

In an empirical project, we only observe x and y and not the realizations of the error term u. In the simulation, we "forget" them and the fact that we know the population parameters and estimate them from our sample using OLS. As motivated in Section 1.9, this will help us to study the behavior of the estimator in a sample like ours.

For our particular sample, the OLS parameter estimates are $\hat{\beta}_0 = 1.09411$ and $\hat{\beta}_1 = 0.49072$. The result of the graph generated in the last lines of Script 2.14 (`SLR-Sim-Sample.jl`) is shown in Figure 2.4. It shows the population regression function with intercept $\beta_0 = 1$ and slope $\beta_1 = 0.5$. It also shows the scatter plot of the sample drawn from this population. This sample led to our OLS regression line with intercept $\hat{\beta}_0 = 1.09411$ and slope $\hat{\beta}_1 = 0.49072$ shown in gray.

Since the SLR assumptions hold in our exercise, Theorems 2.1 and 2.2 of Wooldridge (2019) should apply. Theorem 2.1 implies for our model that the estimators are unbiased, i.e.

$$\mathrm{E}(\hat{\beta}_0) = \beta_0 = 1 \qquad\qquad \mathrm{E}(\hat{\beta}_1) = \beta_1 = 0.5$$

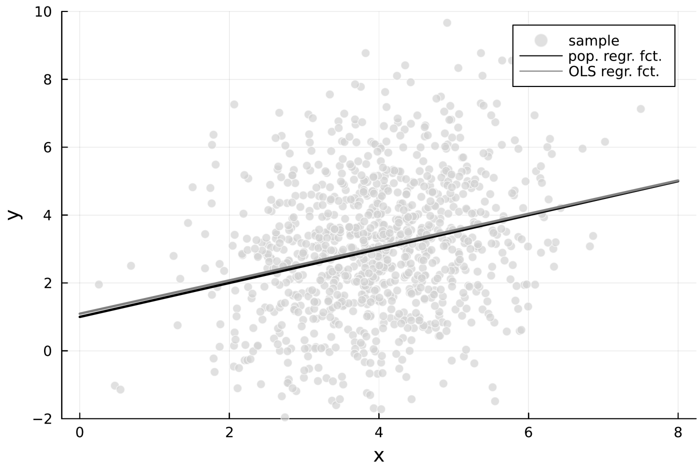
The estimates obtained from our sample are relatively close to their population values. Obviously, we can never expect to hit the population parameter exactly. If we change the random seed by specifying a different number in Script 2.14 (`SLR-Sim-Sample.jl`), we get a different sample and different parameter estimates.

Theorem 2.2 of Wooldridge (2019) states the sampling variance of the estimators conditional on the sample values $\{x_1, \ldots, x_n\}$. It involves the average squared value $\overline{x^2} = 16.856$ and the sum of squares $\sum_{i-1}^{n}(x - \overline{x})^2 = 1069.5$ which we also know from the *Julia* output:

$$\mathrm{Var}(\hat{\beta}_0) = \frac{\sigma^2 \overline{x^2}}{\sum_{i=1}^{n}(x - \overline{x})^2} = \frac{4 \cdot 16.856}{1069.5} = 0.063$$

$$\mathrm{Var}(\hat{\beta}_1) = \frac{\sigma^2}{\sum_{i=1}^{n}(x - \overline{x})^2} = \frac{4}{1069.5} = 0.0037$$

Figure 2.4. Simulated Sample and OLS Regression Line



If Wooldridge (2019) is right, the standard error of $\hat{\beta}_1$ is $\sqrt{0.0037} = 0.0608$. So getting an estimate of $\hat{\beta}_1 = 0.49$ for one sample doesn't seem unreasonable given $\beta_1 = 0.5$.

2.7.2. Many Samples

Since the expected values and variances of our estimators are defined over separate random samples from the same population, it makes sense for us to repeat our simulation exercise over many simulated samples. Just as motivated in Section 1.9, the distribution of OLS parameter estimates across these samples will correspond to the sampling distribution of the estimators.

Script 2.16 (`SLR-Sim-Model-Condx.jl`) implements this with the same **`for`** loop we introduced in Section 1.8.2 and already used for basic Monte Carlo simulations in Section 1.9.1. We analyze $r = 10,000$ samples.

Note that we use the same values for x in all samples since we draw them outside of the loop. We do this to simulate the exact setup of Theorem 2.2 which reports the sampling variances *conditional* on x. In a more realistic setup, we would sample x along with y. The conceptual difference is subtle and the results hardly differ in reasonably large samples. We will come back to these issues in Chapter 5.[2] For each sample, we estimate our parameters and store them in the respective position $i = 1, \ldots, r$ of the arrays **`b0`** and **`b1`**.

Script 2.16: `SLR-Sim-Model-Condx.jl`

```
using Random, GLM, DataFrames, Distributions, Statistics, Plots

# set the random seed:
Random.seed!(12345)

# set sample size and number of simulations:
n = 1000
r = 10000

# set true parameters (betas and sd of u):
beta0 = 1
```

[2] In Script 2.15 (`SLR-Sim-Model.jl`) shown on page 326, we implement the joint sampling from x and y. The results are essentially the same.

```
beta1 = 0.5
su = 2

# initialize b0 and b1 to store results later:
b0 = zeros(r)
b1 = zeros(r)
# draw a sample of x, fixed over replications:
x = rand(Normal(4, 1), n)

# repeat r times:
for i in 1:r
        # draw a sample of y:
        u = rand(Normal(0, su), n)
        y = beta0 .+ beta1 .* x .+ u
        df = DataFrame(y=y, x=x)
        # estimate and store parameters by OLS:
        reg = lm(@formula(y ~ x), df)

        b0[i] = coef(reg)[1]
        b1[i] = coef(reg)[2]
end

# MC estimate of the expected values:
b0_mean = mean(b0)
b1_mean = mean(b1)
println("b0_mean = $b0_mean\n")
println("b1_mean = $b1_mean\n")

# MC estimate of the variances:
b0_var = var(b0)
b1_var = var(b1)
println("b0_var = $b0_var\n")
println("b1_var = $b1_var")

# graph:
x_range = range(0, 8, length=100)

# add population regression line:
plot(x_range, beta0 .+ beta1 .* x_range, ylim=[0, 6],
        color="black", linewidth=2, label="Population")

# add first OLS regression line (to attach a label):
plot!(x_range, b0[1] .+ b1[1] .* x_range,
        color="grey", linewidth=0.5, label="OLS regressions")

# add OLS regression lines no. 2 to 10:
for i in 2:10
        plot!(x_range, b0[i] .+ b1[i] .* x_range,
                color="grey", linewidth=0.5, label=false)
end
ylabel!("y")
xlabel!("x")
savefig("JlGraphs/SLR-Sim-Model-Condx.pdf")
```

Output of Script 2.16: SLR-Sim-Model-Condx.jl

```
b0_mean = 1.0014083956786675

b1_mean = 0.49971507018893335

b0_var = 0.06407053033863686

b1_var = 0.0037976368475450464
```

Script 2.16 (`SLR-Sim-Model-Condx.jl`) gives descriptive statistics of the $r = 10,000$ estimates we got from our simulation exercise. Wooldridge (2019, Theorem 2.1) claims that the OLS estimators are unbiased, so we should expect to get estimates which are very close to the respective population parameters. This is clearly confirmed. The average value of $\hat{\beta}_0$ is very close to $\beta_0 = 1$ and the average value of $\hat{\beta}_1$ is very close to $\beta_1 = 0.5$.

The simulated sampling variances are $\widetilde{\mathrm{Var}}(\hat{\beta}_0) = 0.064$ and $\widetilde{\mathrm{Var}}(\hat{\beta}_1) = 0.0038$. Also these values are very close to the ones we expected from Theorem 2.2. The last lines of the code produce Figure 2.5. It shows the OLS regression lines for the first 10 simulated samples together with the population regression function.

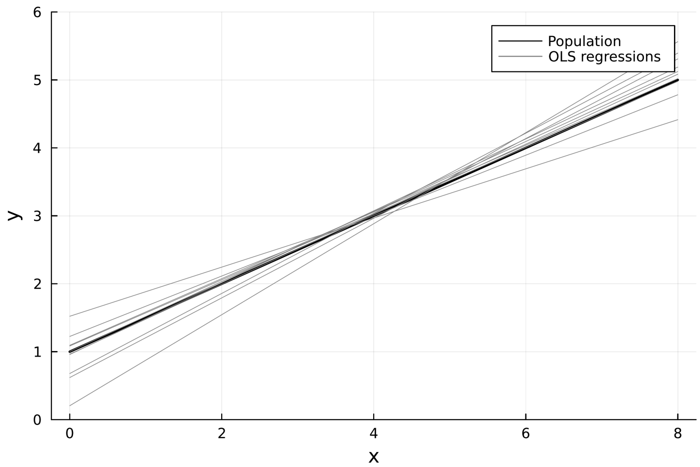
2.7.3. Violation of SLR.4

We will come back to a more systematic discussion of the consequences of violating the SLR assumptions below. At this point, we can already simulate the effects. In order to implement a violation of SLR.4 (zero conditional mean), consider a case where in the population u is not mean independent of x. A simple example is

$$\mathrm{E}(u|x) = \frac{x-4}{5}.$$

What happens to our OLS estimator? Script 2.17 (`SLR-Sim-Model-ViolSLR4.jl`) implements a simulation of this model and is listed in the appendix (p. 327).

Figure 2.5. Population and Simulated OLS Regression Lines



The only line of code we changed compared to Script 2.16 (`SLR-Sim-Model-Condx.jl`) is the sampling of **u** which now reads

```
u_mean = (x .- 4) ./ 5
u = rand.(Normal.(u_mean, su), 1)
```

The simulation results are presented in the output of Script 2.17 (`SLR-Sim-Model-ViolSLR4.jl`). Obviously, the OLS coefficients are now biased: The average estimates are far from the population parameters $\beta_0 = 1$ and $\beta_1 = 0.5$. This confirms that Assumption SLR.4 is required to hold for the unbiasedness shown in Theorem 2.1.

Output of Script 2.17: `SLR-Sim-Model-ViolSLR4.jl`

```
b0_mean = 0.2014083956786684

b1_mean = 0.6997150701889331

b0_var = 0.0640705303386369

b1_var = 0.0037976368475450494
```

2.7.4. Violation of SLR.5

Theorem 2.1 (unbiasedness) does not require Assumption SLR.5 (homoscedasticity), but Theorem 2.2 (sampling variance) does. As an example for a violation consider the population specification

$$\text{Var}(u|x) = \frac{4}{e^{4.5}} \cdot e^x,$$

so SLR.5 is clearly violated since the variance depends on x. We assume exogeneity, so assumption SLR.4 holds. The factor in front ensures that the unconditional variance $\text{Var}(u) = 4$.[3] Based on this unconditional variance only, the sampling variance should not change compared to the results above and we would still expect $\text{Var}(\hat{\beta}_0) = 0.063$ and $\text{Var}(\hat{\beta}_1) = 0.0037$. But since Assumption SLR.5 is violated, Theorem 2.2 is not applicable.

Script 2.18 (`SLR-Sim-Model-ViolSLR5.jl`) implements a simulation of this model and is listed in the appendix (p. 328). Here, we only had to change the line of code for the sampling of **`u`** to

```
u_var = 4 / exp(4.5) .* exp.(x)
u = rand.(Normal.(0, sqrt.(u_var)), 1)
```

The output of Script 2.18 (`SLR-Sim-Model-ViolSLR5.jl`) demonstrates two effects: The unbiasedness provided by Theorem 2.1 is unaffected, but the formula for sampling variance provided by Theorem 2.2 is incorrect.

Output of Script 2.18: `SLR-Sim-Model-ViolSLR5.jl`

```
b0_mean = 1.000195953976725

b1_mean = 0.49992437529387573

b0_var = 0.10866572428292555

b1_var = 0.008591646835322982
```

[3] Since $x \sim \text{Normal}(4, 1)$, e^x is log-normally distributed and has a mean of $e^{4.5}$.

3. Multiple Regression Analysis: Estimation

Running a multiple regression in *Julia* is as straightforward as running a simple regression using the **lm** command in **GLM**. Section 3.1 shows how it is done. Section 3.2 opens the black box and replicates the main calculations using matrix algebra. This is not required for the remaining chapters, so it can be skipped by readers who prefer to keep black boxes closed.

Section 3.3 should not be skipped since it discusses the interpretation of regression results and the prevalent omitted variables problems. Finally, Section 3.4 covers standard errors and multicollinearity for multiple regression.

3.1. Multiple Regression in Practice

Consider the population regression model

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \cdots + \beta_k x_k + u \tag{3.1}$$

and suppose the data set **sample** contains variables **y**, **x1**, **x2**, **x3**, with the respective data of our sample. We estimate the model parameters by OLS using the commands

```
reg = lm(@formula(y ~ x1 + x2 + x3), sample)
```

The tilde "**~**" again separates the dependent variable from the regressors which are now separated using a "**+**" sign. We can add options as before. The constant is again automatically added unless it is explicitly suppressed using **'y ~ 0 + x1 + x2 + x3 + ...'**.

We are already familiar with the workings of **lm**, so the estimation results are stored in a variable **reg**. We can use this variable for further analyses. For a typical regression output including a coefficient table, call **coeftable(reg)**. Further analyses involving residuals, fitted values and the like can be used exactly as presented in Chapter 2.

The output of **coeftable** includes parameter estimates, standard errors according to Theorem 3.2 of Wooldridge (2019), and many more useful results we cannot interpret yet before we have worked through Chapter 4.

Wooldridge, Example 3.1: Determinants of College GPA

This example from Wooldridge (2019) relates the college GPA (colGPA) to the high school GPA (hsGPA) and achievement test score (ACT) for a sample of 141 students. The commands and results can be found in Script 3.1 (Example-3-1.jl). The OLS regression function is

$$\widehat{\texttt{colGPA}} = 1.286 + 0.453 \cdot \texttt{hsGPA} + 0.0094 \cdot \texttt{ACT}.$$

Script 3.1: Example-3-1.jl

```
using WooldridgeDatasets, GLM, DataFrames

gpa1 = DataFrame(wooldridge("gpa1"))

reg = lm(@formula(colGPA ~ hsGPA + ACT), gpa1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg\n")

r2_automatic = r2(reg)
println("r2_automatic = $r2_automatic")
```

Output of Script 3.1: Example-3-1.jl

```
table_reg:

                  Coef.   Std. Error     t  Pr(>|t|)   Lower 95%  Upper 95%

(Intercept)  1.28633      0.340822   3.77    0.0002   0.612419   1.96024
hsGPA        0.453456     0.0958129  4.73    <1e-05   0.264005   0.642907
ACT          0.00942601   0.0107772  0.87    0.3833  -0.0118838  0.0307358


r2_automatic = 0.17642159703480598
```

Wooldridge, Example 3.4: Determinants of College GPA

For the regression run in Example 3.1, the output of Script 3.1 (Example-3-1.jl) reports $R^2 = 0.176$, so about 17.6% of the variance in college GPA is explained by the two regressors.

Examples 3.2, 3.3, 3.5, 3.6: Further Multiple Regression Examples

In order to get a feeling of the methods and results, we present the analyses including the full regression tables of the mentioned Examples from Wooldridge (2019) in Scripts 3.2 (Example-3-2.jl) through 3.6 (Example-3-6.jl). See Wooldridge (2019) for descriptions of the data sets and variables and for comments on the results.

Script 3.2: `Example-3-2.jl`

```
using WooldridgeDatasets, GLM, DataFrames

wage1 = DataFrame(wooldridge("wage1"))

reg = lm(@formula(log(wage) ~ educ + exper + tenure), wage1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Output of Script 3.2: `Example-3-2.jl`

```
table_reg:

                  Coef.   Std. Error      t  Pr(>|t|)    Lower 95%   Upper 95%

(Intercept)  0.28436     0.10419      2.73    0.0066  0.0796756    0.489044
educ         0.092029    0.00732992  12.56    <1e-31  0.0776292    0.106429
exper        0.00412111  0.00172328   2.39    0.0171  0.000735698  0.00750652
tenure       0.0220672   0.00309365   7.13    <1e-11  0.0159897    0.0281447
```

Script 3.3: `Example-3-3.jl`

```
using WooldridgeDatasets, DataFrames, GLM

k401k = DataFrame(wooldridge("401k"))

reg = lm(@formula(prate ~ mrate + age), k401k)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Output of Script 3.3: `Example-3-3.jl`

```
table_reg:

                 Coef.   Std. Error       t  Pr(>|t|)  Lower 95%  Upper 95%

(Intercept)  80.119      0.779021   102.85    <1e-99  78.591     81.6471
mrate         5.52129    0.525884    10.50    <1e-24   4.48976    6.55282
age           0.243147   0.0446999    5.44    <1e-07   0.155467   0.330826
```

Script 3.4: `Example-3-5a.jl`

```
using WooldridgeDatasets, DataFrames, GLM

crime1 = DataFrame(wooldridge("crime1"))

# model without avgsen:
reg = lm(@formula(narr86 ~ pcnv + ptime86 + qemp86), crime1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Output of Script 3.4: `Example-3-5a.jl`

```
table_reg: 

                 Coef.  Std. Error      t  Pr(>|t|)   Lower 95%   Upper 95%

(Intercept)   0.711772   0.0330066   21.56    <1e-94   0.647051    0.776492
pcnv         -0.149927   0.0408653   -3.67    0.0002  -0.230058   -0.0697973
ptime86      -0.0344199  0.008591    -4.01    <1e-04  -0.0512655  -0.0175744
qemp86       -0.104113   0.0103877  -10.02    <1e-22  -0.124482   -0.0837445
```

Script 3.5: `Example-3-5b.jl`

```
using WooldridgeDatasets, DataFrames, GLM

crime1 = DataFrame(wooldridge("crime1"))

# model with avgsen:
reg = lm(@formula(narr86 ~ pcnv + avgsen + ptime86 + qemp86), crime1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Output of Script 3.5: `Example-3-5b.jl`

```
table_reg: 

                  Coef.  Std. Error      t  Pr(>|t|)    Lower 95%   Upper 95%

(Intercept)   0.706756    0.0331515   21.32    <1e-92   0.641752     0.771761
pcnv         -0.150832    0.0408583   -3.69    0.0002  -0.230948    -0.0707154
avgsen        0.00744312  0.00473384   1.57    0.1160  -0.00183918   0.0167254
ptime86      -0.0373908   0.00879407  -4.25    <1e-04  -0.0546345   -0.0201471
qemp86       -0.103341    0.0103965   -9.94    <1e-22  -0.123727    -0.0829552
```

Script 3.6: `Example-3-6.jl`

```
using WooldridgeDatasets, DataFrames, GLM

wage1 = DataFrame(wooldridge("wage1"))

reg = lm(@formula(log(wage) ~ educ), wage1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Output of Script 3.6: `Example-3-6.jl`

```
table_reg: 

                Coef.  Std. Error      t  Pr(>|t|)  Lower 95%  Upper 95%

(Intercept)  0.583773   0.0973358    6.00    <1e-08  0.392556   0.774989
educ         0.0827444  0.00756669  10.94    <1e-24  0.0678796  0.0976091
```

3.2. OLS in Matrix Form

For applying regression methods to empirical problems, we do not actually need to know the formulas our software uses. In multiple regression, we need to resort to matrix algebra in order to find an explicit expression for the OLS parameter estimates. Wooldridge (2019) defers this discussion to Appendix E and we follow the notation used there. Going through this material is not required for applying multiple regression to real-world problems but is useful for a deeper understanding of the methods and their black box implementations in software packages. In the following chapters, we will rely on the comfort of the canned routine **lm**, so this section may be skipped.

In matrix form, we store the regressors in a $n \times (k+1)$ matrix $\mathbf{X}$ which has a column for each regressor plus a column of ones for the constant. The sample values of the dependent variable are stored in a $n \times 1$ column vector $\mathbf{y}$. Wooldridge (2019) derives the OLS estimator $\hat{\boldsymbol{\beta}} = (\hat{\beta}_0, \hat{\beta}_1, \hat{\beta}_2, \ldots, \hat{\beta}_k)'$ to be

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}. \tag{3.2}$$

This equation involves three matrix operations which we know how to implement in *Julia* from Section 1.2.3:

- Transpose: The expression $\mathbf{X}'$ is **transpose(X)**
- Matrix multiplication: The expression $\mathbf{X}'\mathbf{X}$ is translated as **transpose(X) * X**
- Inverse: $(\mathbf{X}'\mathbf{X})^{-1}$ is written as **inv(transpose(X) * X)**

So we can collect everything and translate Equation 3.2 into the somewhat unsightly expression

```
b = inv(transpose(X) * X) * transpose(X) * y
```

The vector of residuals can be manually calculated as

$$\hat{\mathbf{u}} = \mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}} \tag{3.3}$$

or translated into

```
u_hat = y - X * b
```

The formula for the estimated variance of the error term is

$$\hat{\sigma}^2 = \tfrac{1}{n-k-1}\hat{\mathbf{u}}'\hat{\mathbf{u}} \tag{3.4}$$

which is equivalent to

```
sigsq_hat = (transpose(u_hat) * u_hat) / (n - k - 1)
```

The standard error of the regression (SER) is its square root $\hat{\sigma} = \sqrt{\hat{\sigma}^2}$. The estimated OLS variance-covariance matrix according to Wooldridge (2019, Theorem E.2) is then

$$\widehat{\mathrm{Var}(\hat{\boldsymbol{\beta}})} = \hat{\sigma}^2(\mathbf{X}'\mathbf{X})^{-1} \tag{3.5}$$

```
Vbeta_hat = sigsq_hat .* inv(transpose(X) * X)
```

Finally, the standard errors of the parameter estimates are the square roots of the main diagonal of $\text{Var}(\hat{\boldsymbol{\beta}})$ which can be expressed in *Julia* as

```
se = sqrt.(diag(Vbeta_hat))
```

Script 3.7 (`OLS-Matrices.jl`) implements this for the GPA regression from Example 3.1. Comparing the results to the built-in function (see Script 3.1 (`Example-3-1.jl`)), it is reassuring that we get exactly the same numbers for the parameter estimates and standard errors of the coefficients.

Script 3.7: `OLS-Matrices.jl`

```
using WooldridgeDatasets, DataFrames, LinearAlgebra

gpa1 = DataFrame(wooldridge("gpa1"))

# determine sample size & no. of regressors:
n = size(gpa1)[1]
k = 2

# extract y:
y = gpa1.colGPA

# extract X and add a column of ones:
X = hcat(ones(n), gpa1.hsGPA, gpa1.ACT)
# display first rows of X:
X_preview = round.(X[1:3, :], digits=5)
println("X_preview = $X_preview\n")

# parameter estimates:
b = inv(transpose(X) * X) * transpose(X) * y
println("b = $b\n")

# residuals, estimated variance of u and SER:
u_hat = y - X * b
sigsq_hat = (transpose(u_hat) * u_hat) / (n - k - 1)
SER = sqrt(sigsq_hat)
println("SER = $SER\n")

# estimated variance of the parameter estimators and SE:
Vbeta_hat = sigsq_hat .* inv(transpose(X) * X)
se = sqrt.(diag(Vbeta_hat))
println("se = $se")
```

Output of Script 3.7: `OLS-Matrices.jl`

```
X_preview = [1.0 3.0 21.0; 1.0 3.2 24.0; 1.0 3.6 26.0]

b = [1.286327766521133, 0.4534558853481646, 0.009426012260470441]

SER = 0.3403157569643908

se = [0.34082211993697037, 0.09581291608058075, 0.010777187759672879]
```

Script 3.8 (`OLS-Matrices-Formula.jl`) and Script 3.9 (`getMats.jl`) also demonstrates another way of generating **y** and **X** by using the formula syntax to conveniently create all matrices. This is a very useful technique especially in later chapters when using functions that do not support formula syntax.

In Script 3.8 (`OLS-Matrices-Formula.jl`) a formula **`f`** is defined, just as you know it from the **`lm`** command. Note that we do not make use of the **`GLM`** package here, so we have to load the package that provides formula syntax explicitly by

```
using StatsModels
```

It is also worth mentioning that outside from **`GLM`**, we have to include the constant explicitly by adding **`1`** to the formula. Next, we supply this formula and the data to the function **`getMats`**, which is defined in Script 3.9 (`getMats.jl`) and needs to be loaded with **`include`** in Script 3.8 (`OLS-Matrices-Formula.jl`). We use the recommended procedure from the package documentation and will not go into details here.[1] Basically, we apply the formula to the data and extract explanatory and explained variables. We get exactly the same results as in previous examples.

Script 3.8: `OLS-Matrices-Formula.jl`

```
using WooldridgeDatasets, DataFrames, StatsModels, LinearAlgebra
include("getMats.jl")

gpa1 = DataFrame(wooldridge("gpa1"))

# build y and X from a formula:
f = @formula(colGPA ~ 1 + hsGPA + ACT)
xy = getMats(f, gpa1)
y = xy[1]
X = xy[2]

# parameter estimates:
b = inv(transpose(X) * X) * transpose(X) * y
println("b = $b")
```

Output of Script 3.8: `OLS-Matrices-Formula.jl`

```
b = [1.286327766521133, 0.4534558853481646, 0.009426012260470441]
```

Script 3.9: `getMats.jl`

```
# for details, see https://juliastats.org/StatsModels.jl/stable/internals/
using StatsModels

function getMats(formula, df)
    f = apply_schema(formula, schema(formula, df))
    resp, pred = modelcols(f, df)
    return (resp, pred)
end
```

[1]For more information about the package, see Kleinschmidt (2016).

3.3. Ceteris Paribus Interpretation and Omitted Variable Bias

The parameters in a multiple regression can be interpreted as partial effects. In a general model with k regressors, the estimated slope parameter β_j associated with variable x_j is the change of $\hat{y}$ as x_j increases by one unit *and the other variables are held fixed*.

Wooldridge (2019) discusses this interpretation in Section 3.2 and offers a useful formula for interpreting the difference between simple regression results and this *ceteris paribus* interpretation of multiple regression: Consider a regression with two explanatory variables:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2. \tag{3.6}$$

The parameter $\hat{\beta}_1$ is the estimated effect of increasing x_1 by one unit while keeping x_2 fixed. In contrast, consider the simple regression including only x_1 as a regressor:

$$\tilde{y} = \tilde{\beta}_0 + \tilde{\beta}_1 x_1. \tag{3.7}$$

The parameter $\tilde{\beta}_1$ is the estimated effect of increasing x_1 by one unit (and NOT keeping x_2 fixed). It can be related to $\hat{\beta}_1$ using the formula

$$\tilde{\beta}_1 = \hat{\beta}_1 + \hat{\beta}_2 \tilde{\delta}_1 \tag{3.8}$$

where $\tilde{\delta}_1$ is the slope parameter of the linear regression of x_2 on x_1

$$x_2 = \tilde{\delta}_0 + \tilde{\delta}_1 x_1. \tag{3.9}$$

This equation is actually quite intuitive: As x_1 increases by one unit,

- Predicted y directly increases by $\hat{\beta}_1$ units (*ceteris paribus* effect, Equ. 3.6).
- Predicted x_2 increases by $\tilde{\delta}_1$ units (see Equ. 3.9).
- Each of these $\tilde{\delta}_1$ units leads to an increase of predicted y by $\hat{\beta}_2$ units, giving a total indirect effect of $\tilde{\delta}_1 \hat{\beta}_2$ (see again Equ. 3.8)
- The overall effect $\tilde{\beta}_1$ is the sum of the direct and indirect effects (see Equ. 3.8).

We revisit Example 3.1 to see whether we can demonstrate Equation 3.8 in *Julia*. Script 3.10 (`Omitted-Vars.jl`) repeats the regression of the college GPA (`colGPA`) on the achievement test score (`ACT`) and the high school GPA (`hsGPA`). We study the *ceteris paribus* effect of `ACT` on `colGPA` which has an estimated value of $\hat{\beta}_1 = 0.0094$. The estimated effect of `hsGPA` is $\hat{\beta}_2 = 0.453$. The slope parameter of the regression corresponding to Equation 3.9 is $\tilde{\delta}_1 = 0.0389$. Plugging these values into Equation 3.8 gives a total effect of $\tilde{\beta}_1 = 0.0271$ which is exactly what the simple regression at the end of the output delivers.

In this example, the indirect effect is actually stronger than the direct effect. `ACT` predicts `colGPA` mainly because it is related to `hsGPA` which in turn is strongly related to `colGPA`.

These relations hold for the estimates from a given sample. In Section 3.3, Wooldridge (2019) discusses how to apply the same sort of arguments to the OLS estimators which are random variables varying over different samples. Omitting relevant regressors causes bias if we are interested in estimating partial effects. In practice, it is difficult to include *all* relevant regressors making of omitted variables a prevalent problem. It is important enough to have motivated a vast amount of methodological and applied research. More advanced techniques like instrumental variables or panel data methods try to solve the problem in cases where we cannot add all relevant regressors, for example because they are unobservable. We will come back to this in Part 3.

Script 3.10: `Omitted-Vars.jl`

```
using WooldridgeDatasets, DataFrames, GLM

gpa1 = DataFrame(wooldridge("gpa1"))

# parameter estimates for full and simple model:
reg = lm(@formula(colGPA ~ ACT + hsGPA), gpa1)
b = coef(reg)
println("b = $b\n")

# relation between regressors:
reg_delta = lm(@formula(hsGPA ~ ACT), gpa1)
delta_tilde = coef(reg_delta)
println("delta_tilde = $delta_tilde\n")

# omitted variables formula for b1_tilde:
b1_tilde = b[2] + b[3] * delta_tilde[2]
println("b1_tilde = $b1_tilde\n")

# actual regression with hsGPA omitted:
reg_om = lm(@formula(colGPA ~ ACT), gpa1)
b_om = coef(reg_om)
println("b_om = $b_om")
```

Output of Script 3.10: `Omitted-Vars.jl`

```
b = [1.2863277665211537, 0.009426012260472088, 0.4534558853481625]

delta_tilde = [2.4625365832309214, 0.03889675325123465]

b1_tilde = 0.027063973943179713

b_om = [2.4029794730723775, 0.027063973943179418]
```

3.4. Standard Errors, Multicollinearity, and VIF

We have already seen the matrix formula for the conditional variance-covariance matrix under the usual assumptions including homoscedasticity (MLR.5) in Equation 3.5. Theorem 3.2 provides another useful formula for the variance of a single parameter β_j, i.e. for a single element on the main diagonal of the variance-covariance matrix:

$$\mathrm{Var}(\hat{\beta}_j) = \frac{\sigma^2}{SST_j(1-R_j^2)} = \frac{1}{n} \cdot \frac{\sigma^2}{\mathrm{Var}(x_j)} \cdot \frac{1}{1-R_j^2}, \tag{3.10}$$

where $SST_j = \sum_{i=1}^{n}(x_{ji} - \overline{x}_j)^2 = (n-1) \cdot Var(x_j)$ is the total sum of squares and R_j^2 is the usual coefficient of determination from a regression of x_j on all of the other regressors.[2]

[2]Note that here, we use the population variance formula $\mathrm{Var}(x_j) = \frac{1}{n}\sum_{i=1}^{n}(x_{ji} - \overline{x}_j)^2$.

The variance of $\hat{\beta}_j$ consists of four parts:

- $\frac{1}{n}$: The variance is smaller for larger samples.
- σ^2: The variance is larger if the error term varies a lot, since it introduces randomness into the relationship between the variables of interest.
- $\frac{1}{\text{Var}(x_j)}$: The variance is smaller if the regressor x_j varies a lot since this provides relevant information about the relationship.
- $\frac{1}{1-R_j^2}$: This variance inflation factor (VIF) accounts for (imperfect) multicollinearity. If x_j is highly related to the other regressors, R_j^2 and therefore also VIF_j and the variance of $\hat{\beta}_j$ are large.

Since the error variance σ^2 is unknown, we replace it with an estimate to come up with an estimated variance of the parameter estimate. Its square root is the standard error

$$\text{se}(\hat{\beta}_j) = \frac{1}{\sqrt{n}} \cdot \frac{\hat{\sigma}}{\text{sd}(x_j)} \cdot \frac{1}{\sqrt{1-R_j^2}}. \tag{3.11}$$

It is not directly obvious that this formula leads to the same results as the matrix formula in Equation 3.5. We will validate this formula by replicating Example 3.1 which we also used for manually calculating the SE using the matrix formula above. The calculations are shown in Script 3.11 (`MLR-SE.jl`).

Script 3.11: `MLR-SE.jl`

```
using WooldridgeDatasets, DataFrames, GLM, Statistics

gpa1 = DataFrame(wooldridge("gpa1"))

# full estimation results including automatic SE:
reg = lm(@formula(colGPA ~ hsGPA + ACT), gpa1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg\n")

# calculation of SER via residuals:
n = nobs(reg)
k = length(coef(reg))
SER = sqrt(sum(residuals(reg) .^ 2) / (n - k))

# regressing hsGPA on ACT for calculation of R2 & VIF:
reg_hsGPA = lm(@formula(hsGPA ~ ACT), gpa1)
R2_hsGPA = r2(reg_hsGPA)
VIF_hsGPA = 1 / (1 - R2_hsGPA)
println("VIF_hsGPA = $VIF_hsGPA\n")

# manual calculation of SE of hsGPA coefficient:
sdx = std(gpa1.hsGPA) * sqrt((n - 1) / n)
SE_hsGPA = 1 / sqrt(n) * SER / sdx * sqrt(VIF_hsGPA)
println("SE_hsGPA = $SE_hsGPA")
```

Output of Script 3.11: `MLR-SE.jl`

```
table_reg:

                  Coef.  Std. Error     t  Pr(>|t|)   Lower 95%  Upper 95%

(Intercept)  1.28633      0.340822   3.77    0.0002   0.612419   1.96024
hsGPA        0.453456     0.0958129  4.73    <1e-05   0.264005   0.642907
ACT          0.00942601   0.0107772  0.87    0.3833  -0.0118838  0.0307358


VIF_hsGPA = 1.1358234481972784

SE_hsGPA = 0.09581291608057595
```

In Script 3.11 (`MLR-SE.jl`), we extract the SER of the main regression and the R_j^2 from the regression of `hsGPA` on `ACT` which is needed for calculating the VIF for the coefficient of `hsGPA`.[3] **`VIF_hsGPA = 1.1358`** means that the variance of the regression coefficient of `hsGPA` is higher by a factor of (only) 1.1358 than in a world in which it were uncorrelated with the other regressor. The other ingredients of Equation 3.11 are straightforward. The standard error calculated this way is exactly the same as the one of the built-in command and the matrix formula used in Script 3.7 (`OLS-Matrices.jl`).

[3]We could have calculated these values manually like in Scripts 2.8 (`Example-2-8.jl`), 2.13 (`Example-2-12.jl`) or 3.7 (`OLS-Matrices.jl`).

4. Multiple Regression Analysis: Inference

Section 4.1 of Wooldridge (2019) adds assumption MLR.6 (normal distribution of the error term) to the previous assumptions MLR.1 through MLR.5. Together, these assumptions constitute the classical linear model (CLM).

The main additional result we get from this assumption is stated in Theorem 4.1: The OLS parameter estimators are normally distributed (conditional on the regressors $x_1, ..., x_k$). The benefit of this result is that it allows us to do statistical inference similar to the approaches discussed in Section 1.7 for the simple estimator of the mean of a normally distributed random variable.

4.1. The t Test

After the sign and magnitude of the estimated parameters, empirical research typically pays most attention to the results of t tests discussed in this section.

4.1.1. General Setup

An important type of hypotheses we are often interested in is of the form

$$H_0 : \beta_j = a_j, \tag{4.1}$$

where a_j is some given number, very often $a_j = 0$. For the most common case of two-tailed tests, the alternative hypothesis is

$$H_1 : \beta_j \neq a_j, \tag{4.2}$$

and for one-tailed tests it is either one of

$$H_1 : \beta_j < a_j \qquad \text{or} \qquad H_1 : \beta_j > a_j. \tag{4.3}$$

These hypotheses can be conveniently tested using a t test which is based on the test statistic

$$t = \frac{\hat{\beta}_j - a_j}{\text{se}(\hat{\beta}_j)}. \tag{4.4}$$

If H_0 is in fact true and the CLM assumptions hold, then this statistic has a t distribution with $n - k - 1$ degrees of freedom.

4.1.2. Standard Case

Very often, we want to test whether there is any relation at all between the dependent variable y and a regressor x_j and do not want to impose a sign on the partial effect *a priori*. This is a mission for the standard two-sided t test with the hypothetical value $a_j = 0$, so

$$H_0 : \beta_j = 0, \qquad H_1 : \beta_j \neq 0, \tag{4.5}$$

$$t_{\hat{\beta}_j} = \frac{\hat{\beta}_j}{\text{se}(\hat{\beta}_j)}. \tag{4.6}$$

The subscript on the t statistic indicates that this is "the" t value for $\hat{\beta}_j$ for this frequent version of the test. Under H_0, it has the t distribution with $n-k-1$ degrees of freedom implying that the probability that $|t_{\hat{\beta}_j}| > c$ is equal to α if c is the $1-\frac{\alpha}{2}$ quantile of this distribution. If α is our significance level (e.g. $\alpha = 5\%$), then we

$$\text{reject } H_0 \text{ if } |t_{\hat{\beta}_j}| > c$$

in our sample. For the typical significance level $\alpha = 5\%$, the critical value c will be around 2 for reasonably large degrees of freedom and approach the counterpart of 1.96 from the standard normal distribution in very large samples.

The p value indicates the smallest value of the significance level α for which we would still reject H_0 using our sample. So it is the probability for a random variable T with the respective t distribution that $|T| > |t_{\hat{\beta}_j}|$ where $t_{\hat{\beta}_j}$ is the value of the t statistic in our particular sample. In our two-tailed test, it can be calculated as

$$p_{\hat{\beta}_j} = 2 \cdot F_{t_{n-k-1}}(-|t_{\hat{\beta}_j}|), \tag{4.7}$$

where $F_{t_{n-k-1}}(\cdot)$ is the CDF of the t distribution with $n-k-1$ degrees of freedom. If our software provides us with the relevant p values, they are easy to use: We

$$\text{reject } H_0 \text{ if } p_{\hat{\beta}_j} \leq \alpha.$$

Since this standard case of a t test is so common, **`GLM`** provides us with the relevant t and p values directly in the output of **`coeftable`** which we already saw in the previous chapter. The regression table includes for all regressors and the intercept:

- parameter estimates and standard errors, see Section 3.1.
- test statistics $t_{\hat{\beta}_j}$ from Equation 4.6 in the column **`t`**
- respective p values $p_{\hat{\beta}_j}$ from Equation 4.7 in the column **`Pr(>|t|)`**
- respective 95% confidence interval from Equation 4.8 in columns **`Lower 95%`** and **`Upper 95%`** (see Section 4.2)

Wooldridge, Example 4.3: Determinants of College GPA

We have repeatedly used the data set `GPA1` in Chapter 3. This example uses three regressors and estimates a regression model of the form

$$\texttt{colGPA} = \beta_0 + \beta_1 \cdot \texttt{hsGPA} + \beta_2 \cdot \texttt{ACT} + \beta_3 \cdot \texttt{skipped} + u.$$

For the critical values of the t tests, using the normal approximation instead of the exact t distribution with $n-k-1 = 137$ d.f. doesn't make much of a difference:

Script 4.1: `Example-4-3-cv.jl`

```
using Distributions

# CV for alpha=5% and 1% using the t distribution with 137 d.f.:
alpha = [0.05, 0.01]
cv_t = round.(quantile.(TDist(137), 1 .- alpha ./ 2), digits=5)
println("cv_t = $cv_t\n")

# CV for alpha=5% and 1% using the normal approximation:
cv_n = round.(quantile.(Normal(), 1 .- alpha ./ 2), digits=5)
println("cv_n = $cv_n")
```

Output of Script 4.1: `Example-4-3-cv.jl`

```
cv_t = [1.97743, 2.61219]

cv_n = [1.95996, 2.57583]
```

Script 4.2 (`Example-4-3.jl`) presents the output of **`coeftable`** which directly contains all the information to test the hypotheses in Equation 4.5 for all parameters. The t statistics for all coefficients except β_2 are larger in absolute value than the critical value $c = 2.61$ (or $c = 2.58$ using the normal approximation) for $\alpha = 1\%$. So we would reject H_0 for all usual significance levels. By construction, we draw the same conclusions from the p values.
In order to confirm that **`GLM`** is exactly using the formulas of Wooldridge (2019), we next reconstruct the t and p values manually. We extract the coefficients (**`coef`**) and standard errors (**`stderror`**) from the regression results, and simply apply Equations 4.6 and 4.7.

Script 4.2: `Example-4-3.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Distributions

gpa1 = DataFrame(wooldridge("gpa1"))

# store and display results:
reg = lm(@formula(colGPA ~ hsGPA + ACT + skipped), gpa1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg\n")

# manually confirm the formulas, i.e. extract coefficients and SE:
b = coef(reg)
se = stderror(reg)

# reproduce t statistic:
tstat = round.(b ./ se, digits=5)
println("tstat = $tstat\n")

# reproduce p value:
pval = round.(2 * cdf.(TDist(137), -abs.(tstat)), digits=5)
println("pval = $pval")
```

Output of Script 4.2: `Example-4-3.jl`

```
table_reg:

                  Coef.  Std. Error      t  Pr(>|t|)    Lower 95%   Upper 95%

(Intercept)   1.38955     0.331554    4.19    <1e-04   0.73393      2.04518
hsGPA         0.411816    0.0936742   4.40    <1e-04   0.226582     0.59705
ACT           0.0147202   0.0105649   1.39    0.1658  -0.00617107   0.0356115
skipped      -0.0831131   0.0259985  -3.20    0.0017  -0.134523    -0.0317028


tstat = [4.19104, 4.39626, 1.39332, -3.19684]

pval = [5.0e-5, 2.0e-5, 0.16578, 0.00173]
```

4.1.3. Other Hypotheses

For a one-tailed test, the critical value c of the t test and the p values have to be adjusted appropriately. Wooldridge (2019) provides a general discussion in Section 4.2. For testing the null hypothesis $H_0 : \beta_j = a_j$, the tests for the three common alternative hypotheses are summarized in Table 4.1:

Table 4.1. One- and Two-tailed t Tests for $H_0 : \beta_j = a_j$

H_1 :	$\beta_j \neq a_j$	$\beta_j > a_j$	$\beta_j < a_j$
c=quantile	$1 - \frac{\alpha}{2}$	$1 - \alpha$	$1 - \alpha$
reject H_0 if	$\lvert t_{\hat{\beta}_j} \rvert > c$	$t_{\hat{\beta}_j} > c$	$t_{\hat{\beta}_j} < -c$
p value	$2 \cdot F_{t_{n-k-1}}(-\lvert t_{\hat{\beta}_j} \rvert)$	$F_{t_{n-k-1}}(-t_{\hat{\beta}_j})$	$F_{t_{n-k-1}}(t_{\hat{\beta}_j})$

Given the standard regression output like the one in Script 4.2 (`Example-4-3.jl`) including the p value for two-sided tests $p_{\hat{\beta}_j}$, we can easily do one-sided t tests for the null hypothesis $H_0 : \beta_j = 0$ in two steps:

- Is $\hat{\beta}_j$ positive (if $H_1 : \beta_j > 0$) or negative (if $H_1 : \beta_j < 0$)?
  - No $\rightarrow$ Do not reject H_0 since this cannot be evidence against H_0.
  - Yes $\rightarrow$ The relevant p value is half of the reported $p_{\hat{\beta}_j}$.

    $\Rightarrow$ Reject H_0 if $p = \frac{1}{2} p_{\hat{\beta}_j} < \alpha$.

Wooldridge, Example 4.1: Hourly Wage Equation

We have already estimated the wage equation

$$\log(\texttt{wage}) = \beta_0 + \beta_1 \cdot \texttt{educ} + \beta_2 \cdot \texttt{exper} + \beta_3 \cdot \texttt{tenure} + u$$

in Example 3.2. Now we are ready to test $H_0 : \beta_2 = 0$ against $H_1 : \beta_2 > 0$. For the critical values of the t tests, using the normal approximation instead of the exact t distribution with $n - k - 1 = 522$ d.f. doesn't make any relevant difference:

Script 4.3: `Example-4-1-cv.jl`

```
using Distributions

# CV for alpha=5% and 1% using the t distribution with 522 d.f.:
alpha = [0.05, 0.01]
cv_t = round.(quantile.(TDist(522), 1 .- alpha), digits=5)
println("cv_t = $cv_t\n")

# CV for alpha=5% and 1% using the normal approximation:
cv_n = round.(quantile.(Normal(), 1 .- alpha), digits=5)
println("cv_n = $cv_n")
```

Output of Script 4.3: `Example-4-1-cv.jl`

```
cv_t = [1.64778, 2.33351]

cv_n = [1.64485, 2.32635]
```

Script 4.4 (`Example-4-1.jl`) shows the standard regression output. The reported t statistic for the parameter of `exper` is $t_{\hat{\beta}_2} = 2.39$ which is larger than the critical value $c = 2.33$ for the significance level $\alpha = 1\%$, so we reject H_0. By construction, we get the same answer from looking at the p value. Like always, the reported $p_{\hat{\beta}_j}$ value is for a two-sided test, so we have to divide it by 2. The resulting value $p = \frac{0.0171}{2} = 0.00855 < 0.01$, so we reject H_0 using an $\alpha = 1\%$ significance level.

Script 4.4: `Example-4-1.jl`

```
using WooldridgeDatasets, GLM, DataFrames

wage1 = DataFrame(wooldridge("wage1"))

reg = lm(@formula(log(wage) ~ educ + exper + tenure), wage1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Output of Script 4.4: `Example-4-1.jl`

```
table_reg:

                  Coef.   Std. Error      t  Pr(>|t|)    Lower 95%   Upper 95%

(Intercept)  0.28436     0.10419      2.73    0.0066  0.0796756    0.489044
educ         0.092029    0.00732992  12.56    <1e-31  0.0776292    0.106429
exper        0.00412111  0.00172328   2.39    0.0171  0.000735698  0.00750652
tenure       0.0220672   0.00309365   7.13    <1e-11  0.0159897    0.0281447
```

4.2. Confidence Intervals

We have already looked at confidence intervals (CI) for the mean of a normally distributed random variable in Sections 1.7 and 1.9.3. CI for the regression parameters are equally easy to construct and closely related to t tests. Wooldridge (2019, Section 4.3) provides a succinct discussion. The 95% confidence interval for parameter β_j is simply

$$\hat{\beta}_j \pm c \cdot \mathrm{se}(\hat{\beta}_j), \tag{4.8}$$

where c is the same critical value for the two-sided t test using a significance level $\alpha = 5\%$. Wooldridge (2019) shows examples of how to manually construct these CI.

GLM provides the 95% confidence intervals for all parameters in the regression table. In Script 4.5 (`Example-4-8.jl`), we compute other significance levels.

Wooldridge, Example 4.8: Model of R&D Expenditures

We study the relationship between the R&D expenditures of a firm, its size, and the profit margin for a sample of 32 firms in the chemical industry. The regression equation is

$$\log(\texttt{rd}) = \beta_0 + \beta_1 \cdot \log(\texttt{sales}) + \beta_2 \cdot \texttt{profmarg} + u.$$

Script 4.5 (`Example-4-8.jl`) presents the regression results as well as the 95% and 99% CI.

Script 4.5: `Example-4-8.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Distributions

rdchem = DataFrame(wooldridge("rdchem"))

# OLS regression:
reg = lm(@formula(log(rd) ~ log(sales) + profmarg), rdchem)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg\n")

# replicating 95% CI:
alpha = 0.05
CI95_upper = coef(reg) .+ stderror(reg) .* quantile(TDist(32 - 3), alpha / 2)
CI95_lower = coef(reg) .- stderror(reg) .* quantile(TDist(32 - 3), alpha / 2)
println("CI95_upper = $CI95_upper\n")
println("CI95_lower = $CI95_lower\n")

# calculating 99% CI:
alpha = 0.01
CI99_upper = coef(reg) .+ stderror(reg) .* quantile(TDist(32 - 3), alpha / 2)
CI99_lower = coef(reg) .- stderror(reg) .* quantile(TDist(32 - 3), alpha / 2)
println("CI99_upper = $CI99_upper\n")
println("CI99_lower = $CI99_lower")
```

Output of Script 4.5: `Example-4-8.jl`

```
table_reg:

                 Coef.  Std. Error      t  Pr(>|t|)    Lower 95%   Upper 95%

(Intercept)  -4.37827     0.468018   -9.35    <1e-09  -5.33548     -3.42107
log(sales)    1.08422     0.060195   18.01    <1e-16   0.961107     1.20733
profmarg      0.0216557   0.0127826   1.69    0.1010  -0.00448772   0.0477991


CI95_upper = [-5.335478449854266, 0.9611072560097931, -0.004487721638015186]

CI95_lower = [-3.4210680632861976, 1.2073324801318628, 0.04779910329394221]

CI99_upper = [-5.668312696316047, 0.9182992000644498, -0.013578168544388751]

CI99_lower = [-3.0882338168244168, 1.2501405360772062, 0.05688955020031578]
```

4.3. Linear Restrictions: F Tests

Wooldridge (2019, Sections 4.4 and 4.5) discusses more general tests than those for the null hypotheses in Equation 4.1. They can involve one or more hypotheses involving one or more population parameters in a linear fashion.

We follow the illustrative example of Wooldridge (2019, Section 4.5) and analyze major league baseball players' salaries using the data set `MLB1` and the regression model

$$\log(\texttt{salary}) = \beta_0 + \beta_1 \cdot \texttt{years} + \beta_2 \cdot \texttt{gamesyr} + \beta_3 \cdot \texttt{bavg} + \beta_4 \cdot \texttt{hrunsyr} + \beta_5 \cdot \texttt{rbisyr} + u. \tag{4.9}$$

We want to test whether the performance measures batting average (`bavg`), home runs per year (`hrunsyr`), and runs batted in per year (`rbisyr`) have an impact on the salary once we control for the number of years as an active player (`years`) and the number of games played per year (`gamesyr`). So we state our null hypothesis as $H_0 : \beta_3 = 0, \beta_4 = 0, \beta_5 = 0$ versus $H_1 : H_0$ is false, i.e. at least one of the performance measures matters.

The test statistic of the F test is based on the relative difference between the sum of squared residuals in the general (unrestricted) model and a restricted model in which the hypotheses are imposed SSR_{ur} and SSR_r, respectively. In our example, the restricted model is one in which `bavg`, `hrunsyr`, and `rbisyr` are excluded as regressors. If both models involve the same dependent variable, it can also be written in terms of the coefficient of determination in the unrestricted and the restricted model R^2_{ur} and R^2_r, respectively:

$$F = \frac{\text{SSR}_r - \text{SSR}_{ur}}{\text{SSR}_{ur}} \cdot \frac{n-k-1}{q} = \frac{R^2_{ur} - R^2_r}{1 - R^2_{ur}} \cdot \frac{n-k-1}{q}, \tag{4.10}$$

where q is the number of restrictions (in our example, $q = 3$). Intuitively, if the null hypothesis is correct, then imposing it as a restriction will not lead to a significant drop in the model fit and the F test statistic should be relatively small. It can be shown that under the CLM assumptions and the null hypothesis, the statistic has an F distribution with the numerator degrees of freedom equal to q and the denominator degrees of freedom of $n - k - 1$. Given a significance level α, we will reject H_0 if $F > c$, where the critical value c is the $1 - \alpha$ quantile of the relevant $F_{q,n-k-1}$ distribution.

In our example, $n = 353, k = 5, q = 3$. So with $\alpha = 1\%$, the critical value is 3.84 and can be calculated using the **FDist** function in the package **Distributions** as

```
quantile(FDist(3, 347), 1 - 0.01)
```

Script 4.6 (`F-Test.jl`) shows the calculations for this example. The result is F = 9.55 > 3.84, so we clearly reject H_0. We also calculate the p value for this test. It is $p = 4.47 \cdot 10^{-06} = 0.00000447$, so we reject H_0 for any reasonable significance level.

Script 4.6: `F-Test.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Distributions

mlb1 = DataFrame(wooldridge("mlb1"))

# unrestricted OLS regression:
reg_ur = lm(@formula(log(salary) ~
        years + gamesyr + bavg + hrunsyr + rbisyr), mlb1)
r2_ur = r2(reg_ur)
println("r2_ur = $r2_ur\n")

# restricted OLS regression:
reg_r = lm(@formula(log(salary) ~ years + gamesyr), mlb1)
r2_r = r2(reg_r)
println("r2_r = $r2_r\n")

# F statistic:
n = nobs(reg_ur)
fstat = (r2_ur - r2_r) / (1 - r2_ur) * (n - 6) / 3
println("fstat = $fstat\n")

# CV for alpha=1% using the F distribution with 3 and 347 d.f.:
cv = quantile(FDist(3, 347), 1 - 0.01)
println("cv = $cv\n")

# p value = 1-cdf of the appropriate F distribution:
fpval = 1 - cdf(FDist(3, 347), fstat)
println("fpval = $fpval")
```

Output of Script 4.6: `F-Test.jl`

```
r2_ur = 0.6278028485187441

r2_r = 0.5970716339066893

fstat = 9.550253521951943

cv = 3.838520048496029

fpval = 4.473708139829391e-6
```

It should not be surprising that there is a more convenient way to do this. The package **GLM** provides a command **ftest** which is well suited for these kinds of tests. All you have to do, is providing the regression object of the restricted and unrestricted model in that order. We demonstrate the procedure in Script 4.7 (F-Test-Automatic.jl). As in Script 4.6 (F-Test.jl), H_0 is that the three parameters of bavg, hrunsyr, and rbisyr are all equal to zero, so all results are identical to the output of Script 4.6 (F-Test.jl).

Script 4.7: F-Test-Automatic.jl

```
using WooldridgeDatasets, GLM, DataFrames

mlb1 = DataFrame(wooldridge("mlb1"))

# OLS regression:
reg_ur = lm(@formula(log(salary) ~
                years + gamesyr + bavg + hrunsyr + rbisyr), mlb1)
reg_r = lm(@formula(log(salary) ~
                years + gamesyr), mlb1)

# automated F test:
ftest_res = ftest(reg_r.model, reg_ur.model)

fstat = ftest_res.fstat[2]
fpval = ftest_res.pval[2]
println("fstat = $fstat\n")
println("fpval = $fpval")
```

Output of Script 4.7: F-Test-Automatic.jl

```
fstat = 9.550253521951944

fpval = 4.473708139838451e-6
```

The function **ftest** can also be used to test more complicated null hypotheses, although this is much more convenient in *R* or *Python* so far. For example, suppose a sports reporter claims that the batting average plays no role and that the number of home runs has twice the impact as the number of runs batted in. This translates as $H_0 : \beta_3 = 0,\ \beta_4 = 2 \cdot \beta_5$ and gives the following restricted model:

$$\begin{aligned}\log(\texttt{salary}) &= \beta_0 + \beta_1 \cdot \texttt{years} + \beta_2 \cdot \texttt{gamesyr} + 0 \cdot \texttt{bavg} + 2 \cdot \beta_5 \cdot \texttt{hrunsyr} + \beta_5 \cdot \texttt{rbisyr} + u \\ &= \beta_0 + \beta_1 \cdot \texttt{years} + \beta_2 \cdot \texttt{gamesyr} + \beta_5 \cdot (2 \cdot \texttt{hrunsyr} + \texttt{rbisyr}) + u\end{aligned}$$

Equation 4.9 is still our unrestricted model. The output of Script 4.8 (F-Test-Automatic2.jl) shows the results of this test. The p value is $p = 0.6$, so we cannot reject H_0, so the reporter might be right.

Script 4.8: `F-Test-Automatic2.jl`

```
using WooldridgeDatasets, GLM, DataFrames

mlb1 = DataFrame(wooldridge("mlb1"))

# OLS regression:
reg_ur = lm(@formula(log(salary) ~
                years + gamesyr + bavg + hrunsyr + rbisyr), mlb1)

# restrictions "bavg = 0" and "hrunsyr = 2*rbisyr":
mlb1.newvar = 2 * mlb1.hrunsyr + mlb1.rbisyr
reg_r = lm(@formula(log(salary) ~ years + gamesyr + newvar), mlb1)

# automated F test:
ftest_res = ftest(reg_r.model, reg_ur.model)

fstat = ftest_res.fstat[2]
fpval = ftest_res.pval[2]
println("fstat = $fstat\n")
println("fpval = $fpval")
```

Output of Script 4.8: `F-Test-Automatic2.jl`

```
fstat = 0.5117822576247593

fpval = 0.5998780329146316
```

Both the most important and the most straightforward F test is the one for **overall significance**. The null hypothesis is that all parameters except for the constant are equal to zero. If this null hypothesis holds, the regressors do not have any joint explanatory power for y. The results of such a test are easily obtained with the techniques used above. As an example, see Script 4.9 (`Example-4-8-2.jl`). The null hypothesis that neither the sales nor the margin have any relation to R&D spending is clearly rejected with an F statistic of 162.2 and a p value smaller than 10^{-15}.

Script 4.9: `Example-4-8-2.jl`

```
using WooldridgeDatasets, GLM, DataFrames

rdchem = DataFrame(wooldridge("rdchem"))

# OLS regression:
reg_ur = lm(@formula(log(rd) ~ log(sales) + profmarg), rdchem)
reg_r = lm(@formula(log(rd) ~ 1), rdchem)

# automated F test:
ftest_res = ftest(reg_r.model, reg_ur.model)

fstat = ftest_res.fstat[2]
fpval = ftest_res.pval[2]
println("fstat = $fstat\n")
println("fpval = $fpval")
```

Output of Script 4.9: `Example-4-8-2.jl`

```
fstat = 162.23139858478663

fpval = 1.793854336903236e-16
```

4.4. Reporting Regression Results

Now we know most of the statistics shown in a typical regression output. Wooldridge (2019) provides a discussion of how to report them in Section 4.6. We will come back to these issues in more detail in Chapter 19. Here is already a preview of how to conveniently generate tables of different regression results very much like suggested in Wooldridge (2019, Example 4.10).

To automatically generate useful regression tables, we use the package **RegressionTables**.[1] Given multiple regression objects, the command **regtable** generates a table including all of them. We demonstrate this using the following example.

Wooldridge, Example 4.10: Salary-Pension Tradeoff for Teachers

Wooldridge (2019) discusses a model of the tradeoff between salary and pensions for teachers. It boils down to the regression specification

$$\log(\texttt{salary}) = \beta_0 + \beta_1 \cdot (\texttt{benefits/salary}) + \texttt{other factors} + u$$

Script 4.10 (`Example-4-10.jl`) loads the data, generates the new variable `b_s` = (`benefits/salary`) and runs three regressions with different sets of `other factors`. The **regtable** command is then used to display the results in a clearly arranged table of all relevant results.

Script 4.10: **Example-4-10.jl**

```
using WooldridgeDatasets, GLM, DataFrames, RegressionTables

meap93 = DataFrame(wooldridge("meap93"))
meap93.b_s = meap93.benefits ./ meap93.salary

# estimate three different models:
reg1 = lm(@formula(log(salary) ~ b_s), meap93)
reg2 = lm(@formula(log(salary) ~ b_s + log(enroll) + log(staff)), meap93)
reg3 = lm(@formula(log(salary) ~
                b_s + log(enroll) + log(staff) + droprate + gradrate), meap93)

# print results with RegressionTables:
regtable(reg1, reg2, reg3)
```

[1] For more information about the package, see Boehm (2017).

Output of Script 4.10: `Example-4-10.jl`

```
-----------------------------------------------
                         log(salary)
              ---------------------------------
                     (1)          (2)         (3)
-----------------------------------------------
(Intercept)   10.523***    10.844***   10.738***
                 (0.042)      (0.252)     (0.258)
b_s           -0.825***    -0.605***   -0.589***
                 (0.200)      (0.165)     (0.165)
log(enroll)                  0.087***    0.088***
                              (0.007)     (0.007)
log(staff)                  -0.222***   -0.218***
                              (0.050)     (0.050)
droprate                                   -0.000
                                          (0.002)
gradrate                                    0.001
                                          (0.001)
-----------------------------------------------
Estimator            OLS          OLS         OLS
-----------------------------------------------
N                    408          408         408
R2                 0.040        0.353       0.361
-----------------------------------------------
```

5. Multiple Regression Analysis: OLS Asymptotics

Asymptotic theory allows us to relax some assumptions needed to derive the sampling distribution of estimators if the sample size is large enough. For running a regression in a software package, it does not matter whether we rely on stronger assumptions or on asymptotic arguments. So we don't have to learn anything new regarding the implementation.

Instead, this chapter aims to improve on our intuition regarding the workings of asymptotics by looking at some simulation exercises in Section 5.1. Section 5.2 briefly discusses the implementation of the regression-based Lagrange multiplier (LM) test presented by Wooldridge (2019, Section 5.2).

5.1. Simulation Exercises

In Section 2.7, we already used Monte Carlo Simulation methods to study the mean and variance of OLS estimators under the assumptions SLR.1–SLR.5. Here, we will conduct similar experiments but will look at the whole sampling distribution of OLS estimators similar to Section 1.9.2 where we demonstrated the central limit theorem for the sample mean. Remember that the sampling distribution is important since confidence intervals, t and F tests and other tools of inference rely on it.

Theorem 4.1 of Wooldridge (2019) gives the normal distribution of the OLS estimators (conditional on the regressors) based on assumptions MLR.1 through MLR.6. In contrast, Theorem 5.2 states that *asymptotically*, the distribution is normal by assumptions MLR.1 through MLR.5 only. Assumption MLR.6 – the normal distribution of the error terms – is not required if the sample is large enough to justify asymptotic arguments.

In other words: In small samples, the parameter estimates have a normal sampling distribution only if

- the error terms are normally distributed and
- we condition on the regressors.

To see how this works out in practice, we set up a series of simulation experiments. Section 5.1.1 simulates a model consistent with MLR.1 through MLR.6 and keeps the regressors fixed. Theory suggests that the sampling distribution of $\hat{\beta}$ is normal, independent of the sample size. Section 5.1.2 simulates a violation of assumption MLR.6. Normality of $\hat{\beta}$ only holds asymptotically, so for small sample sizes we suspect a violation. Finally, we will look closer into what "conditional on the regressors" means and simulate a (very plausible) violation of this in Section 5.1.3.

5.1.1. Normally Distributed Error Terms

Script 5.1 (`Sim-Asy-OLS-norm.jl`) draws 10,000 samples of a given size (which has to be stored in variable **`n`** before) from a population that is consistent with assumptions MLR.1 through MLR.6. The error terms are specified to be standard normal. The slope estimate $\hat{\beta}_1$ is stored for each of the generated samples in the array **b1**. For a more detailed discussion of the implementation, see Section 2.7.2 where a very similar simulation exercise is introduced.

Script 5.1: Sim-Asy-OLS-norm.jl

```
using Distributions, DataFrames, GLM, Random

# set the random seed:
Random.seed!(12345)

# set sample size and number of simulations:
n = 100
r = 10000
# set true parameters:
beta0 = 1
beta1 = 0.5
sx = 1
ex = 4
# initialize b1 to store results later:
b1 = zeros(r)
# draw a sample of x, fixed over replications:
x = rand(Normal(ex, sx), n)

# repeat r times:
for i in 1:r
    # draw a sample of u (std. normal):
    u = rand(Normal(0, 1), n)
    y = beta0 .+ beta1 .* x .+ u
    df = DataFrame(y=y, x=x)
    # estimate conditional OLS:
    reg = lm(@formula(y ~ x), df)
    b1[i] = coef(reg)[2]
end
```

This code was run for different sample sizes. The density estimate together with the corresponding normal density are shown in Figure 5.1. Not surprisingly, all distributions look very similar to the normal distribution – this is what Theorem 4.1 predicted. Note that the fact that the sampling variance decreases as n rises is only obvious if we pay attention to the different scales of the axes.

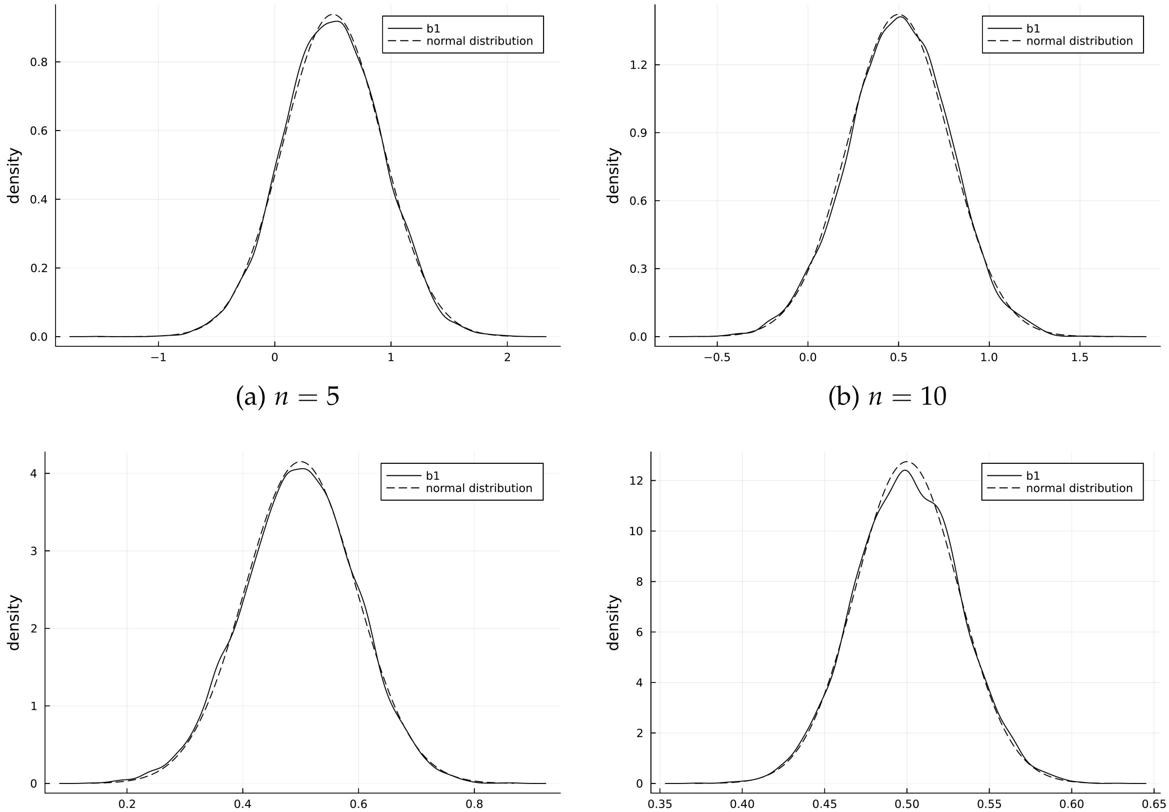
5.1.2. Non-Normal Error Terms

The next step is to simulate a violation of assumption MLR.6. In order to implement a rather drastic violation of the normality assumption similar to Section 1.9.2, we implement a "standardized" χ^2 distribution with one degree of freedom. More specifically, let v be distributed as $\chi^2_{[1]}$. Because this distribution has a mean of 1 and a variance of 2, the error term $u = \frac{v-1}{\sqrt{2}}$ has a mean of 0 and a variance of 1. This simplifies the comparison to the exercise with the standard normal errors above. Figure 5.2 plots the density functions of the standard normal distribution used above and the "standardized" χ^2 distribution. Both have a mean of 0 and a variance of 1 but very different shapes.

Script 5.2 (Sim-Asy-OLS-chisq.jl) implements a simulation of this model and is listed in the appendix (p. 336). The only line of code we changed compared to the previous Script 5.1 (Sim-Asy-OLS-norm.jl) is the sampling of **u** where we replace drawing from a standard normal distribution using **u = rand(Normal(0, 1), n)** with sampling from the standardized $\chi^2_{[1]}$ distribution with

```
u = (rand(Chisq(1), n) .- 1) ./ sqrt(2)
```

Figure 5.1. Density of $\hat{\beta}_1$ with Different Sample Sizes: Normal Error Terms



(a) $n = 5$ (b) $n = 10$

(c) $n = 100$ (d) $n = 1\,000$

Figure 5.2. Density Functions of the Simulated Error Terms

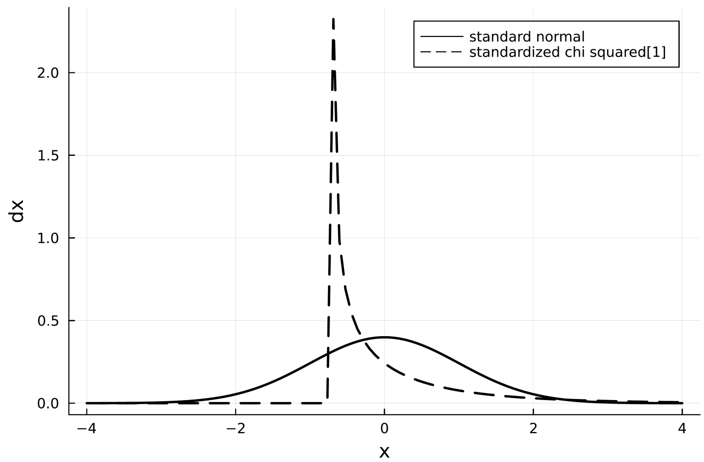
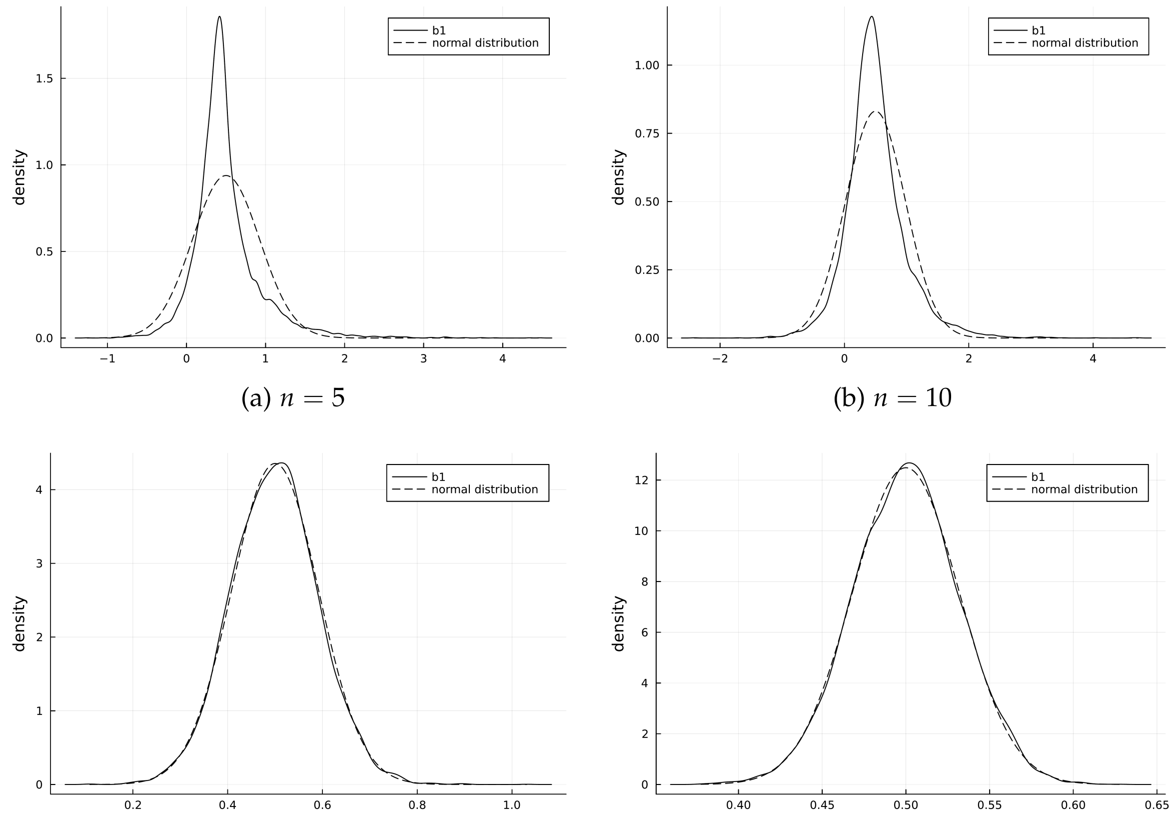


Figure 5.3. Density of $\hat{\beta}_1$ with Different Sample Sizes: Non-Normal Error Terms



(a) $n = 5$

(b) $n = 10$

(c) $n = 100$

(d) $n = 1\,000$

For each of the same sample sizes used above, we again estimate the slope parameter for 10,000 samples. The densities of $\hat{\beta}_1$ are plotted in Figure 5.3 together with the respective normal distributions with the corresponding variances. For the small sample sizes, the deviation from the normal distribution is strong. Note that the dashed normal distributions have the same mean and variance. The main difference is the kurtosis which is larger than 8 in the simulations for $n = 5$ compared to the normal distribution for which the kurtosis is equal to 3.

For larger sample sizes, the sampling distribution of $\hat{\beta}_1$ converges to the normal distribution. For $n = 10$, the difference is smaller but still discernible. For $n = 1\,000$, it cannot be detected anymore in our simulation exercise. How large the sample needs to be depends among other things on the severity of the violations of MLR.6. If the distribution of the error terms is not as extremely non-normal as in our simulations, smaller sample sizes like the rule of thumb $n = 30$ might suffice for valid asymptotics.

5.1.3. (Not) Conditioning on the Regressors

There is a more subtle difference between the finite-sample results regarding the variance (Theorem 3.2) and distribution (Theorem 4.1) on one hand and the corresponding asymptotic results (Theorem 5.2). The former results describe the sampling distribution "conditional on the sample values of the independent variables". This implies that as we draw different samples, the values of the regressors $x_1, \ldots, x_k$ remain the same and only the error terms and dependent variables change.

In our previous simulation exercises in Scripts like 2.16 (`SLR-Sim-Model-Condx.jl`), 5.1 (`Sim-Asy-OLS-norm.jl`), and 5.2 (`Sim-Asy-OLS-chisq.jl`), this is implemented by making random draws of x outside of the simulation loop. This is a realistic description of how data is generated only in some simple experiments: The experimenter chooses the regressors for the sample, conducts the experiment and measures the dependent variable.

In most applications we are concerned with, this is an unrealistic description of how we obtain our data. If we draw a sample of individuals, both their dependent and independent variables differ across samples. In these cases, the distribution "conditional on the sample values of the independent variables" can only serve as an approximation of the actual distribution with varying regressors. For large samples, this distinction is irrelevant and the asymptotic distribution is the same.

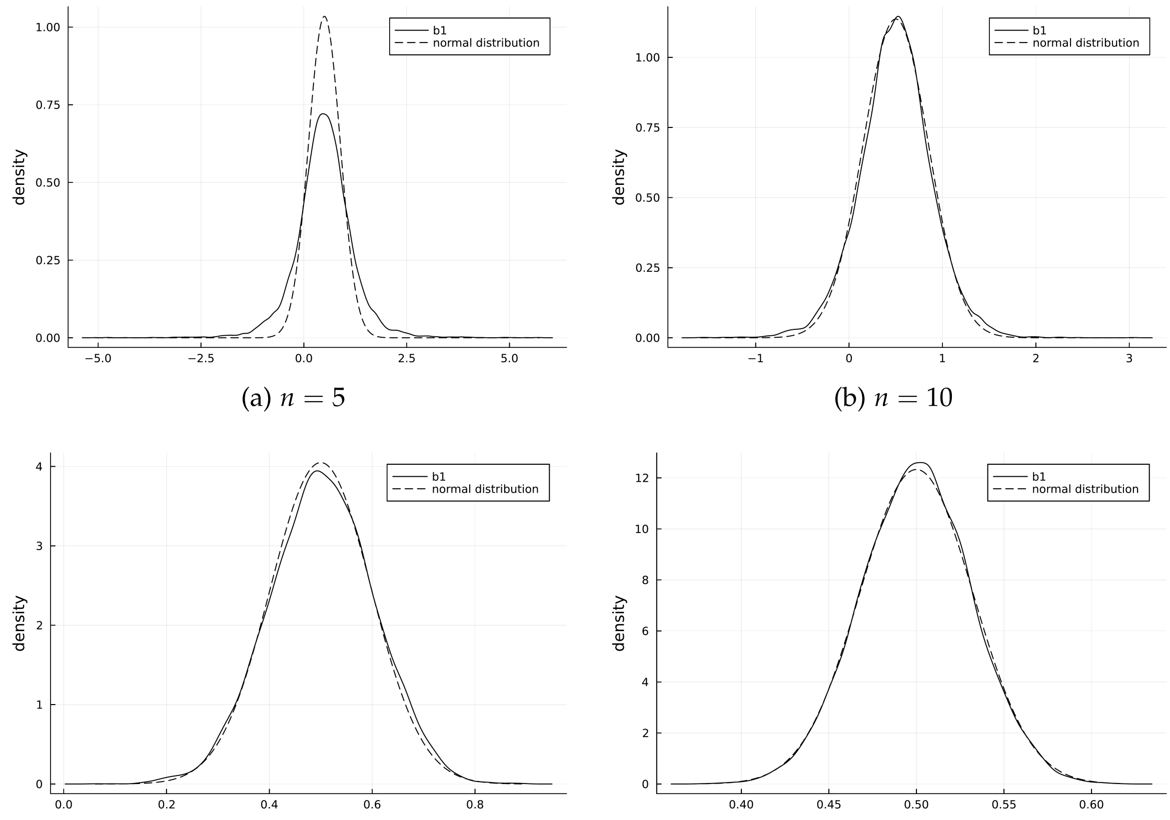
Let's see how this plays out in an example. Script 5.3 (`Sim-Asy-OLS-uncond.jl`) differs from Script 5.1 (`Sim-Asy-OLS-norm.jl`) only by moving the generation of the regressors into the loop in which the 10,000 samples are generated. This is inconsistent with Theorem 4.1, so for small samples, we don't know the distribution of $\hat{\beta}_1$. Theorem 5.2 is applicable, so for (very) large samples, we know that the estimator is normally distributed.

Figure 5.4 shows the distribution of the 10,000 estimates generated by Script 5.3 (`Sim-Asy-OLS-uncond.jl`) for $n = 5, 10, 100$, and 1 000. As we expected from theory, the distribution is (close to) normal for large samples. For small samples, it deviates quite a bit. The kurtosis is 10.25 for a sample size of $n = 5$ which is far away from the kurtosis of 3 of a normal distribution.

Script 5.3: `Sim-Asy-OLS-uncond.jl`

```
using Distributions, DataFrames, GLM, Random

# set the random seed:
Random.seed!(12345)

# set sample size and number of simulations:
n = 100
r = 10000

# set true parameters:
beta0 = 1
beta1 = 0.5
sx = 1
ex = 4

# initialize b1 to store results later:
b1 = zeros(r)

# repeat r times:
for i in 1:r
    # draw a sample of x, varying over replications:
    x = rand(Normal(ex, sx), n)
    # draw a sample of u (std. normal):
    u = rand(Normal(0, 1), n)
    y = beta0 .+ beta1 .* x .+ u
    df = DataFrame(y=y, x=x)
    # estimate unconditional OLS:
    reg = lm(@formula(y ~ x), df)
    b1[i] = coef(reg)[2]
end
```

Figure 5.4. Density of $\hat{\beta}_1$ with Different Sample Sizes: Varying Regressors



(a) $n = 5$ (b) $n = 10$

(c) $n = 100$ (d) $n = 1\,000$

5.2. LM Test

As an alternative to the F tests discussed in Section 4.3, LM tests for the same sort of hypotheses can be very useful with large samples. In the linear regression setup, the test statistic is

$$LM = n \cdot R^2_{\tilde{u}},$$

where n is the sample size and $R^2_{\tilde{u}}$ is the usual R^2 statistic in a regression of the residual $\tilde{u}$ from the restricted model on the unrestricted set of regressors. Under the null hypothesis, it is asymptotically distributed as χ^2_q with q denoting the number of restrictions. Details are given in Wooldridge (2019, Section 5.2).

The implementation in **`GLM`** is straightforward if we remember that the residuals can be obtained with the **`residuals`** function.

Wooldridge, Example 5.3: Economic Model of Crime

We analyze the same data on the number of arrests as in Example 3.5. The unrestricted regression model equation is

$$\texttt{narr86} = \beta_0 + \beta_1\texttt{pcnv} + \beta_2\texttt{avgsen} + \beta_3\texttt{tottime} + \beta_4\texttt{ptime86} + \beta_5\texttt{qemp86} + u.$$

The dependent variable `narr86` reflects the number of times a man was arrested and is explained by the proportion of prior arrests (`pcnv`), previous average sentences (`avgsen`), the time spend in prison before 1986 (`tottime`), the number of months in prison in 1986 (`ptime86`), and the number of quarters unemployed in 1986 (`qemp86`).
The joint null hypothesis is

$$H_0 : \beta_2 = \beta_3 = 0,$$

so the restricted set of regressors excludes `avgsen` and `tottime`. Script 5.4 (`Example-5-3.jl`) shows an implementation of this LM test. The restricted model is estimated and its residuals **`utilde`**$=\tilde{u}$ are calculated. They are regressed on the unrestricted set of regressors. The R^2 from this regression is 0.001494, so the LM test statistic is calculated to be around $LM = 0.001494 \cdot 2725 = 4.071$. This is smaller than the critical value for a significance level of $\alpha = 10\%$, so we do not reject the null hypothesis. We can also easily calculate the p value using the χ^2 CDF. It turns out to be 0.1306.
The same hypothesis can be tested using the F test presented in Section 4.3 using the command **`ftest`**. In this example, it delivers the same p value up to three digits.

Script 5.4: `Example-5-3.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Distributions

crime1 = DataFrame(wooldridge("crime1"))

# 1. estimate restricted model:
reg_r = lm(@formula(narr86 ~ pcnv + ptime86 + qemp86), crime1)
r2_r = r2(reg_r)
println("r2_r = $r2_r\n")

# 2. regression of residuals from restricted model:
crime1.utilde = residuals(reg_r)
reg_LM = lm(@formula(utilde ~
                pcnv + ptime86 + qemp86 + avgsen + tottime), crime1)
r2_LM = r2(reg_LM)
println("r2_LM = $r2_LM\n")

# 3. calculation of LM test statistic:
LM = r2_LM * nobs(reg_LM)
println("LM = $LM\n")

# 4. critical value from chi-squared distribution, alpha=10%:
cv = quantile(Chisq(2), 1 - 0.10)
println("cv = $cv\n")

# 5. p value (alternative to critical value):
pval = 1 - cdf(Chisq(2), LM)
println("pval = $pval\n")

# 6. compare to F test:
reg_ur = lm(@formula(narr86 ~
                pcnv + ptime86 + qemp86 + avgsen + tottime), crime1)
# hypotheses: "avgsen = 0" and "tottime = 0"
reg_r = lm(@formula(narr86 ~ pcnv + ptime86 + qemp86), crime1)
ftest_res = ftest(reg_r.model, reg_ur.model)

fstat = ftest_res.fstat[2]
fpval = ftest_res.pval[2]
println("fstat = $fstat\n")
println("fpval = $fpval")
```

Output of Script 5.4: `Example-5-3.jl`

```
r2_r = 0.041323307701239265

r2_LM = 0.0014938456737831896

LM = 4.070729461059192

cv = 4.605170185988092

pval = 0.13063282803348197

fstat = 2.0339215584291725

fpval = 0.13102048172866032
```

6. Multiple Regression Analysis: Further Issues

In this chapter, we cover some issues regarding the implementation of regression analyses. Section 6.1 discusses more flexible specification of regression equations such as variable scaling, standardization, polynomials and interactions. They can be conveniently included in the **formula** and used in the **GLM** OLS estimation. Section 6.2 is concerned with predictions and their confidence and prediction intervals.

6.1. Model Formulae

If we run a regression in **GLM** using a syntax like

```
lm(@formula(y ~ x1 + x2 + x3), sample)
```

the expression **y ~ x1 + x2 + x3** is referred to as a model **formula**. It is a compact symbolic way to describe our regression equation. The dependent variable is separated from the regressors by a "**~**" and the regressors are separated by a "**+**" indicating that they enter the equation in a linear fashion. A constant is added by default by the **lm** command. Such formulae can be specified in more complex ways to indicate different kinds of regression equations. We will cover the most important ones in this section.

6.1.1. Data Scaling: Arithmetic Operations Within a Formula

Wooldridge (2019) discusses how different scaling of the variables in the model affects the parameter estimates and other statistics in Section 6.1. As an example, we consider a model relating the birth weight to cigarette smoking of the mother during pregnancy and the family income. The basic model equation is

$$\texttt{bwght} = \beta_0 + \beta_1 \texttt{cigs} + \beta_2 \texttt{faminc} + u \tag{6.1}$$

which translates into formula syntax as **bwght ~ cigs + faminc**.

If we want to measure the weight in pounds rather than ounces, there are two ways to implement different rescaling in *Julia*. We can

- Define a different variable like **bwght_lbs = bwght ./ 16** and use this variable in the formula: **bwght_lbs ~ cigs + faminc**
- Specify this rescaling directly in the formula: **(bwght/16) ~ cigs + faminc**

Note that **+** and ***** have a formula-specific meaning, so some arithmetic operations are problematic within the formula. In these cases, the function **identity** disables formula syntax in parts of the formula. To convert from ounces back to pounds by **bwght_lbs * 16** in the formula, for example, we use **identity** in Script 6.1 (`Data-Scaling.jl`). If we want to measure the number of cigarettes smoked per day in packs, we could again define a new variable **packs = cigs ./ 20** and use it as a regressor or simply specify the formula **bwght ~ (cigs/20) + faminc**. Brackets are not mandatory, but increase the readability of the formula.

Script 6.1 (`Data-Scaling.jl`) demonstrates these features. As discussed in Wooldridge (2019, Section 6.1), dividing the dependent variable by 16 changes all coefficients by the same factor $\frac{1}{16}$ and dividing a regressor by 20 changes its coefficient by the factor 20. Other statistics like R^2 are unaffected.

Script 6.1: Data-Scaling.jl

```
using WooldridgeDatasets, GLM, DataFrames, RegressionTables

bwght = DataFrame(wooldridge("bwght"))

# regress and report coefficients:
reg = lm(@formula(bwght ~ cigs + faminc), bwght)

# weight in pounds, manual way:
bwght.bwght_lbs = bwght.bwght ./ 16
reg_lbs1 = lm(@formula(bwght_lbs ~ cigs + faminc), bwght)

# weight in pounds, direct way:
reg_lbs2 = lm(@formula((bwght / 16) ~ cigs + faminc), bwght)

# packs of cigaretts:
reg_packs = lm(@formula(bwght ~ (cigs / 20) + faminc), bwght)

# weight in ounces using bwght_lbs:
reg_pds = lm(@formula(identity(bwght_lbs * 16) ~ cigs + faminc), bwght)

# print results with RegressionTables:
regtable(reg, reg_lbs1, reg_lbs2, reg_packs, reg_pds)
```

Output of Script 6.1: Data-Scaling.jl

```
-----------------------------------------------------------------------------------------
                    bwght     bwght_lbs    bwght / 16       bwght    identity(bwght_lbs * 16)
               ----------    ---------    ----------   ----------    ------------------------
                      (1)          (2)           (3)          (4)                         (5)
-----------------------------------------------------------------------------------------
(Intercept)    116.974***     7.311***      7.311***   116.974***                  116.974***
                  (1.049)      (0.066)       (0.066)      (1.049)                     (1.049)
cigs            -0.463***    -0.029***     -0.029***                                -0.463***
                  (0.092)      (0.006)       (0.006)                                  (0.092)
faminc            0.093**      0.006**       0.006**      0.093**                     0.093**
                  (0.029)      (0.002)       (0.002)      (0.029)                     (0.029)
cigs / 20                                                -9.268***
                                                          (1.832)
-----------------------------------------------------------------------------------------
Estimator             OLS          OLS           OLS          OLS                         OLS
-----------------------------------------------------------------------------------------
N                   1,388        1,388         1,388        1,388                       1,388
R2                  0.030        0.030         0.030        0.030                       0.030
-----------------------------------------------------------------------------------------
```

6.1.2. Standardization: Beta Coefficients

A specific arithmetic operation is the standardization. A variable is standardized by subtracting its mean and dividing by its standard deviation. For example, the standardized dependent variable y

and regressor x_1 are

$$z_y = \frac{y - \bar{y}}{\mathrm{sd}(y)} \quad \text{and} \quad z_{x_1} = \frac{x_1 - \bar{x}_1}{\mathrm{sd}(x_1)}. \tag{6.2}$$

If the regression model only contains standardized variables, the coefficients have a special interpretation. They measure by how many *standard deviations* y changes as the respective independent variable increases by *one standard deviation*. Inconsistent with the notation used here, they are sometimes referred to as beta coefficients.

In *Julia*, we can use the same type of arithmetic transformations as in Section 6.1.1 to subtract the mean and divide by the standard deviation. It can be done more conveniently by defining and using a function **scale** directly for all variables we want to standardize. Defining a function was introduced in Section 1.8.3 and Script 6.2 (`Example-6-1.jl`) demonstrates the use of **scale** in the context of a regression.

Wooldridge, Example 6.1: Effects of Pollution on Housing Prices

We are interested in how air pollution (`nox`) and other neighborhood characteristics affect the value of a house. A model using standardization for all variables is expressed in a formula as

price_sc ~ 0 + nox_sc + crime_sc + rooms_sc + dist_sc + stratio_sc

with **variable_sc** denoting the scaled version of **variable**. The output of Script 6.2 (`Example-6-1.jl`) shows the parameter estimates of this model. The house price drops by 0.34 standard deviations as the air pollution increases by one standard deviation.

Script 6.2: `Example-6-1.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Statistics

hprice2 = DataFrame(wooldridge("hprice2"))

# define a function for the standardization:
function scale(x)
    x_mean = mean(x)
    x_var = var(x)
    x_scaled = (x .- x_mean) ./ sqrt.(x_var)
    return x_scaled
end

# standardize and estimate:
hprice2.price_sc = scale(hprice2.price)
hprice2.nox_sc = scale(hprice2.nox)
hprice2.crime_sc = scale(hprice2.crime)
hprice2.rooms_sc = scale(hprice2.rooms)
hprice2.dist_sc = scale(hprice2.dist)
hprice2.stratio_sc = scale(hprice2.stratio)

reg = lm(@formula(price_sc ~
        0 + nox_sc + crime_sc + rooms_sc + dist_sc + stratio_sc),
    hprice2)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Output of Script 6.2: `Example-6-1.jl`

```
table_reg:

                Coef.  Std. Error      t  Pr(>|t|)  Lower 95%   Upper 95%

nox_sc      -0.340446   0.0444966  -7.65    <1e-12  -0.427869  -0.253023
crime_sc    -0.143283   0.0306861  -4.67    <1e-05  -0.203572  -0.0829934
rooms_sc     0.513888   0.0300002  17.13    <1e-51   0.454946   0.57283
dist_sc     -0.234839   0.0429788  -5.46    <1e-07  -0.319279  -0.150398
stratio_sc  -0.27028    0.0299399  -9.03    <1e-17  -0.329103  -0.211457
```

6.1.3. Logarithms

We have already seen in Section 2.4 that we can include the function **`log`** directly in formulas to represent logarithmic and semi-logarithmic models. A simple example of a partially logarithmic model and its formula would be

$$\log(y) = \beta_0 + \beta_1 \log(x_1) + \beta_2 x_2 + u \tag{6.3}$$

which can be expressed as **`log(y) ~ log(x1) + x2`**.

Script 6.3 (`Formula-Logarithm.jl`) shows this again for the house price example. As the air pollution `nox` increases by *one percent*, the house price drops by about 0.72 *percent*. As the number of rooms increases by *one*, the value of the house increases by roughly 30.6%. Wooldridge (2019, Section 6.2) discusses how the latter value is only an approximation and the actual estimated effect is $(\exp(0.306) - 1) = 0.358$ which is 35.8%.

Script 6.3: `Formula-Logarithm.jl`

```
using WooldridgeDatasets, GLM, DataFrames

hprice2 = DataFrame(wooldridge("hprice2"))

reg = lm(@formula(log(price) ~ log(nox) + rooms), hprice2)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Output of Script 6.3: `Formula-Logarithm.jl`

```
table_reg:

                 Coef.  Std. Error       t  Pr(>|t|)  Lower 95%  Upper 95%

(Intercept)   9.23374    0.187741    49.18    <1e-99   8.86489    9.60259
log(nox)     -0.717674   0.0663397  -10.82    <1e-23  -0.848011  -0.587337
rooms         0.305918   0.0190174   16.09    <1e-46   0.268555   0.343282
```

6.1.4. Quadratics and Polynomials

Specifying quadratic terms or higher powers of regressors can be a useful way to make a model more flexible by allowing the partial effects or (semi-)elasticities to decrease or increase with the value of the regressor.

Instead of creating additional variables containing the squared value of a regressor, in *Julia* we can simply add **`x^2`** to a formula. Higher order terms are specified accordingly. A simple cubic model and its corresponding formula are

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + u \tag{6.4}$$

which translates to **`y ~ x + x^2 + x^3`** in formula syntax.

For nonlinear models like this, it is often useful to get a graphical illustration of the effects. Section 6.2.2 shows how to conveniently generate these.

Wooldridge, Example 6.2: Effects of Pollution on Housing Prices

This example of Wooldridge (2019) demonstrates the combination of logarithmic and quadratic specifications. The model for house prices is

$$\log(\texttt{price}) = \beta_0 + \beta_1 \log(\texttt{nox}) + \beta_2 \log(\texttt{dist}) + \beta_3 \texttt{rooms} + \beta_4 \texttt{rooms}^2 + \beta_5 \texttt{stratio} + u.$$

Script 6.4 (`Example-6-2.jl`) implements this model and presents detailed results including t statistics and their p values. The quadratic term of `rooms` has a significantly positive coefficient $\hat{\beta}_4$ implying that the semi-elasticity increases with more rooms. The negative coefficient for `rooms` and the positive coefficient for $\texttt{rooms}^2$ imply that for "small" numbers of rooms, the price *decreases* with the number of rooms and for "large" values, it *increases*. The number of rooms implying the smallest price can be found as[1]

$$\texttt{rooms}^* = \frac{-\beta_3}{2\beta_4} \approx 4.4.$$

Script 6.4: `Example-6-2.jl`

```
using WooldridgeDatasets, GLM, DataFrames

hprice2 = DataFrame(wooldridge("hprice2"))

reg = lm(@formula(log(price) ~
                log(nox) + log(dist) + rooms + (rooms^2) + stratio), hprice2)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Output of Script 6.4: `Example-6-2.jl`

```
table_reg:

                  Coef.  Std. Error      t  Pr(>|t|)  Lower 95%    Upper 95%

(Intercept)  13.3855     0.566473    23.63    <1e-82  12.2725    14.4984
log(nox)     -0.901682   0.114687    -7.86    <1e-13  -1.12701   -0.676354
log(dist)    -0.0867813  0.0432807   -2.01    0.0455  -0.171816  -0.00174687
rooms        -0.545113   0.165454    -3.29    0.0011  -0.870184  -0.220042
rooms ^ 2     0.0622612  0.012805     4.86    <1e-05   0.037103   0.0874194
stratio      -0.0475902  0.00585419  -8.13    <1e-14  -0.059092  -0.0360884
```

[1] We need to find $\texttt{rooms}^*$ to minimize $\beta_3 \texttt{rooms} + \beta_4 \texttt{rooms}^2$. Setting the first derivative $\beta_3 + 2\beta_4 \texttt{rooms}$ equal to zero and solving for `rooms` delivers the result.

6.1.5. Hypothesis Testing

A natural question to ask is whether a regressor has additional statistically significant explanatory power in a regression model, given all the other regressors. In simple model specifications, this question can be answered by a simple t test, so the results for all regressors are available with a quick look at the standard regression table.[2] When working with polynomials or other specifications, the influence of one regressor is captured by several parameters. We can test its significance with an F test of the joint null hypothesis that all of these parameters are equal to zero. As an example, let's revisit Example 6.2:

$$\log(\texttt{price}) = \beta_0 + \beta_1 \log(\texttt{nox}) + \beta_2 \log(\texttt{dist}) + \beta_3 \texttt{rooms} + \beta_4 \texttt{rooms}^2 + \beta_5 \texttt{stratio} + u$$

The significance of `rooms` can be assessed with an F test of $H_0 : \beta_3 = \beta_4 = 0$. As discussed in Section 4.3, such a test can be performed with the command **ftest** from the package **GLM**. This is shown in Script 6.5 (`Example-6-2-Ftest.jl`).

Script 6.5: `Example-6-2-Ftest.jl`

```
using WooldridgeDatasets, GLM, DataFrames

hprice2 = DataFrame(wooldridge("hprice2"))

reg_ur = lm(@formula(log(price) ~
                log(nox) + log(dist) + rooms + (rooms^2) + stratio), hprice2)

# testing hypotheses rooms = 0 and rooms^2 = 0:
reg_r = lm(@formula(log(price) ~
                log(nox) + log(dist) + stratio), hprice2)

ftest_res = ftest(reg_r.model, reg_ur.model)
fstat = ftest_res.fstat[2]
fpval = ftest_res.pval[2]
println("fstat = $fstat\n")
println("fpval = $fpval")
```

Output of Script 6.5: `Example-6-2-Ftest.jl`

```
fstat = 110.41878192669476

fpval = 1.91932500195311e-40
```

6.1.6. Interaction Terms

Models with interaction terms allow the effect of one variable x_1 to depend on the value of another variable x_2. A simple model including an interaction term would be

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2 + u. \qquad (6.5)$$

Of course, we can implement this in *Julia* by defining a new variable containing the product of the two regressors. But again, a direct specification in the model formula is more convenient. The expression **x1 & x2** within a formula adds the interaction term $x_1 x_2$. Even more conveniently, **x1 * x2** adds not only the interaction but also both original variables allowing for a very concise syntax. So the model in Equation 6.5 can be specified in *Julia* as either of the two formulas:

[2] Section 4.1 discusses t tests.

$$\texttt{y ~ x1 + x2 + x1 \& x2} \quad \Leftrightarrow \quad \texttt{y ~ x1 * x2}$$

If one variable x_1 is interacted with a set of other variables, they can be grouped by parentheses to allow for a compact syntax. For example, the shortest way to express the model equation

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_1 x_2 + \beta_5 x_1 x_3 + u \tag{6.6}$$

in *Julia* syntax is `y ~ x1 * (x2 + x3)`.

Wooldridge, Example 6.3: Effects of Attendance on Final Exam Performance

This example analyzes a model including a standardized dependent variable, quadratic terms and an interaction. Standardized scores in the final exam are explained by class attendance, prior performance and an interaction term:

$$\texttt{stndfnl} = \beta_0 + \beta_1 \texttt{atndrte} + \beta_2 \texttt{priGPA} + \beta_3 \texttt{ACT} + \beta_4 \texttt{priGPA}^2 + \beta_5 \texttt{ACT}^2 + \beta_6 \texttt{priGPA} \cdot \texttt{atndrte} + u$$

Script 6.6 (`Example-6-3.jl`) estimates this model.
The effect of attending classes is

$$\frac{\partial \texttt{stndfnl}}{\partial \texttt{atndrte}} = \beta_1 + \beta_6 \texttt{priGPA}.$$

For the average $\overline{\texttt{priGPA}} = 2.59$, the script estimates this partial effect to be around 0.0078. It tests the null hypothesis that this effect is zero using an F test by plugging in $\beta_1 + \beta_6 \cdot 2.59 = 0 \Leftrightarrow \beta_1 = -\beta_6 \cdot 2.59$, which gives the following restricted model:

$$\texttt{stndfnl} = \beta_0 - \beta_6 \cdot 2.59 \cdot \texttt{atndrte} + \beta_2 \texttt{priGPA} + \beta_3 \texttt{ACT} + \beta_4 \texttt{priGPA}^2 + \beta_5 \texttt{ACT}^2 + \beta_6 \texttt{priGPA} \cdot \texttt{atndrte} + u$$

$$\Leftrightarrow$$

$$\texttt{stndfnl} = \beta_0 + \beta_6(-2.59 \cdot \texttt{atndrte} + \texttt{priGPA} \cdot \texttt{atndrte}) + \beta_2 \texttt{priGPA} + \beta_3 \texttt{ACT} + \beta_4 \texttt{priGPA}^2 + \beta_5 \texttt{ACT}^2 + u$$

With a p value of 0.0034, this hypothesis can be rejected at all common significance levels.

Script 6.6: `Example-6-3.jl`

```
using WooldridgeDatasets, GLM, DataFrames

attend = DataFrame(wooldridge("attend"))

reg_ur = lm(@formula(stndfnl ~ atndrte * priGPA + ACT +
                               (priGPA^2) + (ACT^2)), attend)
table_reg_ur = coeftable(reg_ur)
println("table_reg_ur: \n$table_reg_ur\n")

# estimate for partial effect at priGPA=2.59:
b = coef(reg_ur)
partial_effect = b[2] + 2.59 * b[7]
println("partial_effect = $partial_effect\n")

# F test for partial effect at priGPA=2.59:
attend.pe = -2.59 .* attend.atndrte .+ attend.atndrte .* attend.priGPA
reg_r = lm(@formula(stndfnl ~ pe + priGPA + ACT +
                              (priGPA^2) + (ACT^2)), attend)
ftest_res = ftest(reg_r.model, reg_ur.model)
fstat = ftest_res.fstat[2]
fpval = ftest_res.pval[2]
println("fstat = $fstat\n")
println("fpval = $fpval")
```

Output of Script 6.6: `Example-6-3.jl`

```
table_reg_ur:

                     Coef.   Std. Error      t  Pr(>|t|)     Lower 95%    Upper 95%

(Intercept)         2.05029    1.36032      1.51    0.1322  -0.620686      4.72127
atndrte            -0.00671293 0.0102321   -0.66    0.5120  -0.0268036     0.0133777
priGPA             -1.62854    0.481003    -3.39    0.0008  -2.57299      -0.684094
ACT                -0.128039   0.098492    -1.30    0.1940  -0.321428      0.0653492
priGPA ^ 2          0.295905   0.101049     2.93    0.0035   0.0974944     0.494315
ACT ^ 2             0.00453336 0.00217642   2.08    0.0376   0.000259961   0.00880676
atndrte & priGPA    0.00558591 0.00431739   1.29    0.1962  -0.00289126    0.0140631


partial_effect = 0.007754572228611521

fstat = 8.63258105674091

fpval = 0.003414992399585733
```

6.2. Prediction

In this section, we are concerned with predicting the value of the dependent variable y given certain values of the regressors $x_1, \ldots, x_k$. If these are the regressor values in our estimation sample, we called these predictions "fitted values" and discussed their calculation in Section 2.2. Now, we generalize this to arbitrary values and add standard errors, confidence intervals, and prediction intervals.

6.2.1. Confidence and Prediction Intervals for Predictions

Confidence intervals reflect the uncertainty about the *expected value* of the dependent variable given values of the regressors. If we are interested in predicting the college GPA of an *individual*, prediction intervals account for the additional uncertainty regarding the unobserved characteristics reflected by the error term u.

Given a model

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_k x_k + u \tag{6.7}$$

we are interested in the expected value of y given the regressors take specific values $c_1, c_2, \ldots, c_k$:

$$\theta_0 = \mathrm{E}(y|x_1 = c_1, \ldots, x_k = c_k) = \beta_0 + \beta_1 c_1 + \beta_2 c_2 + \cdots + \beta_k c_k. \tag{6.8}$$

The natural point estimates are

$$\hat{\theta}_0 = \hat{\beta}_0 + \hat{\beta}_1 c_1 + \hat{\beta}_2 c_2 + \cdots + \hat{\beta}_k c_k \tag{6.9}$$

and can readily be obtained once the parameter estimates $\hat{\beta}_0, \ldots, \hat{\beta}_k$ are calculated.

Standard errors and confidence intervals are less straightforward to compute. Wooldridge (2019, Section 6.4) suggests a smart way to obtain these from a modified regression. **`GLM`** provides an even simpler and more convenient approach.

The function **`predict`** automatically calculates $\hat{\theta}_0$. The function can be called on an object created by the **`lm`** function. Its argument is a data frame containing the values of the regressors $c_1, \ldots c_k$ of the regressors $x_1, \ldots x_k$ with the same variable names as in the data frame used for estimation. If we don't have one yet, it can for example be specified as

```
DataFrame(id="newobservation1", x1 = c1, x2 = c2, ... , xk = ck)
```

where **x1** through **xk** are the variable names and **c1** through **ck** are the values which can also be specified as vectors to get predictions at several values of the regressors. See Section 1.2.4 for more on data frames and Script 6.7 (`Predictions.jl`) for an example.

Script 6.7: `Predictions.jl`

```
using WooldridgeDatasets, GLM, DataFrames

gpa2 = DataFrame(wooldridge("gpa2"))

reg = lm(@formula(colgpa ~ sat + hsperc + hsize + (hsize^2)), gpa2)

# print regression table:
table_reg = coeftable(reg)
println("table_reg: \n$table_reg\n")

# generate data set containing the regressor values for predictions:
cvalues1 = DataFrame(id="newPerson1", sat=1200, hsperc=30, hsize=5)
println("cvalues1: \n$cvalues1\n")

# point estimate of prediction (cvalues1):
colgpa_pred1 = round.(predict(reg, cvalues1), digits=5)
println("colgpa_pred1 = $colgpa_pred1\n")

# define three sets of regressor variables:
cvalues2 = DataFrame(id=["newPerson1", "newPerson2", "newPerson3"],
    sat=[1200, 900, 1400],
    hsperc=[30, 20, 5], hsize=[5, 3, 1])
println("cvalues2: \n$cvalues2\n")

# point estimate of prediction (cvalues2):
colgpa_pred2 = round.(predict(reg, cvalues2), digits=5)
println("colgpa_pred2 = $colgpa_pred2")
```

Output of Script 6.7: `Predictions.jl`

```
table_reg:

                  Coef.   Std. Error       t  Pr(>|t|)    Lower 95%    Upper 95%

(Intercept)   1.49265    0.0753414     19.81    <1e-82   1.34494      1.64036
sat           0.0014925  6.52134e-5    22.89    <1e-99   0.00136464   0.00162035
hsperc       -0.0138558  0.000561005  -24.70    <1e-99  -0.0149557   -0.0127559
hsize        -0.0608815  0.0165012     -3.69    0.0002  -0.0932328   -0.0285302
hsize ^ 2     0.0054603  0.00226985     2.41    0.0162   0.00101017   0.00991042


cvalues1:
1×4 DataFrame
 Row | id          sat    hsperc  hsize
     | String      Int64  Int64   Int64
     ---------------------------------------------
   1 | newPerson1   1200      30      5

colgpa_pred1 = [2.70008]

cvalues2:
3×4 DataFrame
```

```
 Row | id          sat    hsperc  hsize
     | String      Int64  Int64   Int64
     ---------------------------------------------
   1 | newPerson1   1200      30      5
   2 | newPerson2    900      20      3
   3 | newPerson3   1400       5      1

colgpa_pred2 = [2.70008, 2.42528, 3.45745]
```

The function **predict** calculates not only $\hat{\theta}_0$, but also

- confidence intervals, if the argument **interval=:confidence** is used and
- prediction intervals, if the argument **interval=:prediction** is used. Wooldridge (2019) explains how to calculate the prediction interval manually.

Script 6.8 (Example-6-5.jl) demonstrates the procedure for $\alpha = 5\%$ and 1%.

Wooldridge, Example 6.5: Confidence Interval for Predicted College GPA

We try to predict the college GPA, for example to support the admission decisions for our college. Our regression model equation is

$$\texttt{colgpa} = \beta_0 + \beta_1 \texttt{sat} + \beta_2 \texttt{hsperc} + \beta_3 \texttt{hsize} + \beta_4 \texttt{hsize}^2 + u.$$

Script 6.8 (Example-6-5.jl) shows the implementation of the estimation and prediction. The estimation results are stored as the variable **reg**. The values of the regressors for which we want to do the prediction are stored in the new data frame **cvalues2**. Then the function **predict** is called multiple times with different arguments. For an SAT score of 1200, a high school percentile of 30 and a high school size of 5 (i.e. 500 students), the predicted college GPA is 2.7. Wooldridge (2019) obtains the same value using a general but more cumbersome regression approach. We define two other types of students with different values of sat, hsperc, and hsize in the data frame **cvalues2**.
Script 6.8 (Example-6-5.jl) also calculates the 95% and 99% confidence and prediction intervals. The object **colgpa_CI_95** contains the 95% confidence interval, for example, which is reported in columns **lower** and **upper**. With 95% confidence we can say that the expected college GPA for students with the features of the student named **newPerson1** is between 2.66 and 2.74. The object **colgpa_PI_99** contains the 99% prediction interval, for example, which is also reported in columns **lower** and **upper**. All results are the same as those manually calculated by Wooldridge (2019).

Script 6.8: Example-6-5.jl

```
using WooldridgeDatasets, GLM, DataFrames

gpa2 = DataFrame(wooldridge("gpa2"))

reg = lm(@formula(colgpa ~ sat + hsperc + hsize + (hsize^2)), gpa2)

# define three sets of regressor variables:
cvalues2 = DataFrame(
    id=["newPerson1", "newPerson2", "newPerson3"],
    sat=[1200, 900, 1400],
    hsperc=[30, 20, 5],
    hsize=[5, 3, 1])

# point estimates and 95% confidence and prediction intervals:
colgpa_CI_95 = predict(reg, cvalues2, interval=:confidence)
```

```
println("colgpa_CI_95: \n$colgpa_CI_95\n")
colgpa_PI_95 = predict(reg, cvalues2, interval=:prediction)
println("colgpa_PI_95: \n$colgpa_PI_95\n")

# point estimates and 99% confidence and prediction intervals:
colgpa_CI_99 = predict(reg, cvalues2, interval=:confidence, level=0.99)
println("colgpa_CI_99: \n$colgpa_CI_99\n")
colgpa_PI_99 = predict(reg, cvalues2, interval=:prediction, level=0.99)
println("colgpa_PI_99: \n$colgpa_PI_99")
```

Output of Script 6.8: `Example-6-5.jl`

```
colgpa_CI_95:
3×3 DataFrame
 Row | prediction  lower     upper
     | Float64?    Float64?  Float64?
    ---------------------------------------------
   1 |    2.70008   2.6611    2.73905
   2 |    2.42528   2.39733   2.45324
   3 |    3.45745   3.40277   3.51213

colgpa_PI_95:
3×3 DataFrame
 Row | prediction  lower     upper
     | Float64?    Float64?  Float64?
    ---------------------------------------------
   1 |    2.70008   1.60175   3.7984
   2 |    2.42528   1.32729   3.52327
   3 |    3.45745   2.35845   4.55644

colgpa_CI_99:
3×3 DataFrame
 Row | prediction  lower     upper
     | Float64?    Float64?  Float64?
    ---------------------------------------------
   1 |    2.70008   2.64885   2.7513
   2 |    2.42528   2.38854   2.46203
   3 |    3.45745   3.38557   3.52932

colgpa_PI_99:
3×3 DataFrame
 Row | prediction  lower     upper
     | Float64?    Float64?  Float64?
    ---------------------------------------------
   1 |    2.70008  1.25639    4.14376
   2 |    2.42528  0.982034   3.86853
   3 |    3.45745  2.01288    4.90202
```

6.2.2. Effect Plots for Nonlinear Specifications

In models with quadratic or other nonlinear terms, the coefficients themselves are often difficult to interpret directly. We have to do additional calculations to obtain the partial effect at different values of the regressors or derive the extreme points. In Example 6.2, we found the number of rooms implying the minimum predicted house price to be around 4.4.

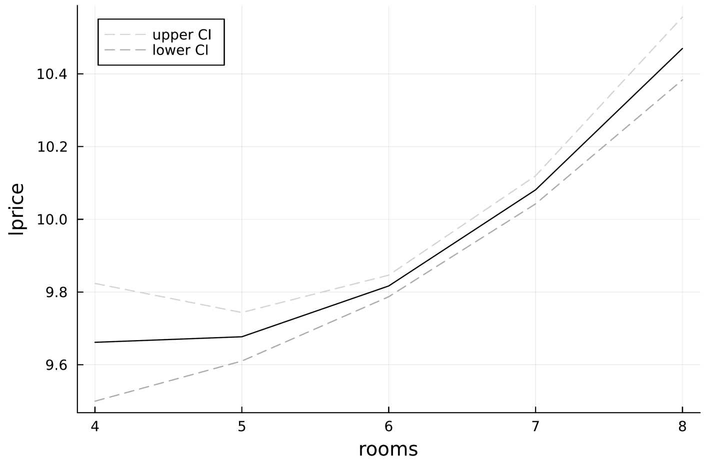
For a better visual understanding of the implications of our model, it is often useful to calculate predictions for *different values of one regressor* of interest while keeping *the other regressors fixed* at certain values like their overall sample means. By plotting the results against the regressor value, we get a very intuitive graph showing the estimated *ceteris paribus* effects of the regressor.

We already know how to calculate predictions and their confidence intervals from Section 6.2.1. Script 6.9 (`Effects-Manual.jl`) repeats the regression from Example 6.2 and creates an effects plot for the number of rooms. The number of rooms is varied between 4 and 8 and the other variables are set to their respective sample means for all predictions. The regressor values and the implied predictions are shown in a table and then plotted with their confidence bands. We see the minimum at a number of rooms of around 4. The resulting graph is shown in Figure 6.1.

Script 6.9: `Effects-Manual.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Plots, Statistics

hprice2 = DataFrame(wooldridge("hprice2"))

# repeating the regression from Example 6.2:
reg = lm(@formula(log(price) ~
        log(nox) + log(dist) + rooms + (rooms^2) + stratio), hprice2)

# predictions with rooms = 4-8, all others at the sample mean:
nox_mean = mean(hprice2.nox)
dist_mean = mean(hprice2.dist)
stratio_mean = mean(hprice2.stratio)
X = DataFrame(
    rooms=4:8,
    nox=nox_mean,
    dist=dist_mean,
    stratio=stratio_mean)
println("X: \n$X\n")

# calculate 95% confidence interval:
lpr_CI = predict(reg, X, interval=:confidence)
println("lpr_CI: \n$lpr_CI\n")

# plot:
plot(X.rooms, lpr_CI.prediction, color="black", label=false, legend=:topleft)
plot!(X.rooms, lpr_CI.upper, color="lightgrey", linestyle=:dash, label="upper CI")
plot!(X.rooms, lpr_CI.lower, color="darkgrey", linestyle=:dash, label="lower CI")
ylabel!("lprice")
xlabel!("rooms")
savefig("JlGraphs/Effects-Manual.pdf")
```

Figure 6.1. Nonlinear Effects in Example 6.2



Output of Script 6.9: `Effects-Manual.jl`

```
X:
5×4 DataFrame
 Row | rooms  nox      dist     stratio
     | Int64  Float64  Float64  Float64
     ---------------------------------------------
   1 |     4  5.54978  3.79575  18.4593
   2 |     5  5.54978  3.79575  18.4593
   3 |     6  5.54978  3.79575  18.4593
   4 |     7  5.54978  3.79575  18.4593
   5 |     8  5.54978  3.79575  18.4593

lpr_CI:
5×3 DataFrame
 Row | prediction  lower     upper
     | Float64?    Float64?  Float64?
     ---------------------------------------------
   1 |    9.6617    9.49981   9.82359
   2 |    9.67694   9.61021   9.74367
   3 |    9.8167    9.78706   9.84635
   4 |   10.081    10.0424   10.1196
   5 |   10.4698   10.3834   10.5562
```

7. Multiple Regression Analysis with Qualitative Regressors

Many variables of interest are qualitative rather than quantitative. Examples include gender, race, labor market status, marital status, and brand choice. In this chapter, we discuss the use of qualitative variables as regressors. Wooldridge (2019, Section 7.5) also covers linear probability models with a binary dependent variable in a linear regression. Since this does not change the implementation, we will skip this topic here and cover binary dependent variables in Chapter 17.

Qualitative information can be represented as binary or dummy variables which can only take the value zero or one. In Section 7.1, we see that dummy variables can be used as regressors just as any other variable. An even more natural way to store yes/no type of information in *Julia* is to use Boolean variables which can also be directly used as regressors, see Section 7.2.

While qualitative variables with more than two outcomes can be represented by a set of dummy variables, the more natural and convenient way to do this are categorical variables as covered in Section 1.2.4. A special case in which we wish to break a numeric variable into categories is discussed in Section 7.4. Finally, Section 7.5 revisits interaction effects and shows how these can be used with categorical variables to conveniently allow and test for difference in the regression equation.

7.1. Linear Regression with Dummy Variables as Regressors

If qualitative data are stored as dummy variables (i.e. variables taking the values zero or one), these can easily be used as regressors in linear regression. If a single dummy variable is used in a model, its coefficient represents the difference in the intercept between groups, see Wooldridge (2019, Section 7.2).

A qualitative variable can also take $g > 2$ values. A variable `MobileOS` could for example take one of the $g = 4$ values “Android”, “iOS”, “Windows”, or “other”. This information can be represented by $g - 1$ dummy variables, each taking the values zero or one, where one category is left out to serve as a reference category. They take the value one if the respective operating system is used and zero otherwise. Wooldridge (2019, Section 7.3) gives more information on these variables and their interpretation.

Here, we are concerned with implementing linear regressions with dummy variables as regressors. Everything works as before once we have generated the dummy variables. In the example data sets provided with Wooldridge (2019), this has usually already been done for us, so we don’t have to learn anything new in terms of implementation. We show two examples.

Wooldridge, Example 7.1: Hourly Wage Equation

We are interested in the wage differences by gender and regress the hourly wage on a dummy variable which is equal to one for females and zero for males. We also include regressors for education, experience, and tenure. The implementation with **GLM** is standard and the dummy variable **female** is used just as any other regressor as shown in Script 7.1 (`Example-7-1.jl`). Its estimated coefficient of -1.81 indicates that on average, a woman makes \$1.81 per hour less than a man *with the same education, experience, and tenure.*

Script 7.1: Example-7-1.jl

```
using WooldridgeDatasets, GLM, DataFrames

wage1 = DataFrame(wooldridge("wage1"))

reg = lm(@formula(wage ~ female + educ + exper + tenure), wage1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Output of Script 7.1: Example-7-1.jl

```
table_reg:
                  Coef.  Std. Error      t  Pr(>|t|)    Lower 95%   Upper 95%

(Intercept)  -1.56794     0.724551   -2.16    0.0309  -2.99134     -0.144538
female       -1.81085     0.264825   -6.84    <1e-10  -2.33111     -1.2906
educ          0.571505    0.0493373  11.58    <1e-27   0.47458      0.668429
exper         0.0253959   0.0115694   2.20    0.0286   0.00266739   0.0481243
tenure        0.141005    0.0211617   6.66    <1e-10   0.0994323    0.182578
```

Wooldridge, Example 7.6: Log Hourly Wage Equation

We used log wage as the dependent variable and distinguish gender and marital status using a qualitative variable with the four outcomes "single female", "single male", "married female", and "married male". We actually implement this regression using an interaction term between **married** and **female** in Script 7.2 (`Example-7-6.jl`). *Relative to the reference group* of single males *with the same education, experience, and tenure*, married males make about 21.3% more (the coefficient of **married**), and single females make about 11.0% less (the coefficient of **female**). The coefficient of the interaction term implies that married females make around 30.1%-21.3%=8.7% less *than single females*, 30.1%+11.0%=41.1% less than married males, and 30.1%+11.0%-21.3%=19.8% less than single males. Note once again that the approximate interpretation as percent may be inaccurate, see Section 6.1.3.

Script 7.2: `Example-7-6.jl`

```
using WooldridgeDatasets, GLM, DataFrames

wage1 = DataFrame(wooldridge("wage1"))

reg = lm(@formula(log(wage) ~
        married * female + educ + exper + (exper^2) +
        tenure + (tenure^2)), wage1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Output of Script 7.2: `Example-7-6.jl`

```
table_reg:

                        Coef.   Std. Error      t  Pr(>|t|)     Lower 95%     Upper 95%

(Intercept)        0.321378     0.100009      3.21    0.0014   0.124904      0.517852
married            0.212676     0.0553572     3.84    0.0001   0.103923      0.321428
female            -0.11035      0.0557421    -1.98    0.0483  -0.219859     -0.000841431
educ               0.0789103    0.0066945    11.79    <1e-27   0.0657585     0.092062
exper              0.0268006    0.00524285    5.11    <1e-06   0.0165007     0.0371005
exper ^ 2         -0.000535245  0.000110426  -4.85    <1e-05  -0.000752184  -0.000318307
tenure             0.0290875    0.006762      4.30    <1e-04   0.0158031     0.0423719
tenure ^ 2        -0.000533143  0.000231243  -2.31    0.0215  -0.000987434  -7.88514e-5
married & female  -0.300593     0.0717669    -4.19    <1e-04  -0.441584     -0.159602
```

7.2. Boolean Variables

A natural way for storing qualitative yes/no information in *Julia* is to use Boolean variables introduced in Section 1.2.2. They can take the values **true** or **false** and can be transformed into a 0/1 dummy variable with the function **Int** where **true=1** and **false=0**. 0/1-coded dummies can *vice versa* be transformed into logical variables with the function **Bool**.

Instead of transforming Boolean variables into dummies, they can be directly used as regressors. For this, the argument **contrasts** in the **lm** function must be set as

```
lm(@formula(y ~ x), sample, contrasts=Dict(:varname => DummyCoding()))
```

to generate a dummy from a variable named **varname**.

The coefficient is then named **varname: true** indicating that **true** was treated as **1**. Script 7.3 (Example-7-1-Boolean.jl) repeats the analysis of Example 7.1 with the regressor **female** being coded as **true** or **false** instead of a 0/1 dummy variable.

Script 7.3: Example-7-1-Boolean.jl

```
using WooldridgeDatasets, GLM, DataFrames

wage1 = DataFrame(wooldridge("wage1"))

# regression with boolean variable:
wage1.isfemale = Bool.(wage1.female)
reg = lm(@formula(wage ~ isfemale + educ + exper + tenure), wage1,
    contrasts=Dict(:isfemale => DummyCoding()))

table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Output of Script 7.3: Example-7-1-Boolean.jl

```
table_reg:

                    Coef.  Std. Error      t  Pr(>|t|)    Lower 95%   Upper 95%

(Intercept)     -1.56794     0.724551   -2.16    0.0309  -2.99134     -0.144538
isfemale: true  -1.81085     0.264825   -6.84    <1e-10  -2.33111     -1.2906
educ             0.571505    0.0493373  11.58    <1e-27   0.47458      0.668429
exper            0.0253959   0.0115694   2.20    0.0286   0.00266739   0.0481243
tenure           0.141005    0.0211617   6.66    <1e-10   0.0994323    0.182578
```

In real-world data sets, qualitative information is often not readily coded as logical or dummy variables, so we might want to create our own regressors. Suppose a qualitative variable saved as the array **OS** takes one of the three string values "Android", "iOS", "Windows", or "other". We can manually define the three relevant logical variables with "Android" as the reference category with

```
iOS  = OS .== "iOS"
wind = OS .== "Windows"
oth  = OS .== "other"
```

A more convenient and elegant way to deal with qualitative variables are categorical variables discussed in the next section.

7.3. Categorical Variables

We have introduced categorical variables in Section 1.2.4. They take one of a given set of outcomes, so they are well suited to store qualitative information.

In a linear regression performed by **GLM** we can easily transform any variable into a categorical variable using the **contrasts** argument from the previous section. The function **lm** is clever enough to implicitly add $g-1$ dummy variables if the variable has g outcomes. As a reference category, the first category is left out by default. For string variables this even works without specifying the **contrasts** argument, because the package implementing the formula syntax treats such variables as categorical variables and uses them as dummys.

Script 7.4 (Regr-Categorical.jl) shows how categorical variables are used. It uses the data set CPS1985.[1] This data set is similar to the one used in Examples 7.1 and 7.6 in that it contains wage and other data for 534 individuals. The frequency tables for the two variables **gender** and **occupation** are shown in the output. The variable **gender** has two categories **male** and **female**. The variable **occupation** has six categories.

In the output, the coefficients are labeled with a combination of the variable and category name. As an example, the estimated coefficient of 0.224 for **gender: male** in **results** implies that men make about 22.4% more than women who are the same in terms of the other regressors. Employees in technical positions earn around 1% (see coefficient of **oc: technical**) less than otherwise equal management positions (who are the reference category).

We can choose different reference categories using the argument **base** of the function **DummyCoding**, where we provide a new reference group **somegroup** with the command **:varname => DummyCoding(base="somegroup")**. In the specification **reg_newref**, we choose **male** and **technical**. When we rerun the same regression command, we see the expected results: Variables like **education** and **experience** get the same coefficients. The dummy variable for females gets the negative of what the males got previously. Obviously, it is equivalent to say "female log wages are lower by 0.224" and "male log wages are higher by 0.224".

The coefficients for the occupation are now relative to **technical**. From the first regression we already knew that technical positions make 1% less than managers, so it is not surprising that in the second regression we find that managers make 1% more than technical positions. The other occupation coefficients are higher by 0.010085 implying the same relative comparisons as in the first specification.

Script 7.4: Regr-Categorical.jl

```
using WooldridgeDatasets, GLM, DataFrames, FreqTables, CSV

CPS1985 = DataFrame(CSV.File("data/CPS1985.csv"))
# rename variable to make outputs more compact:
rename!(CPS1985, :occupation => :oc)

# table of categories and frequencies for two categorical variables:
freq_gender = freqtable(CPS1985.gender)
println("freq_gender: \n$freq_gender\n")

freq_occupation = freqtable(CPS1985.oc)
println("freq_occupation: \n$freq_occupation\n")

# directly using categorical variables in regression formula
# (the formula automatically interprets string
# columns as categorical variables and dummy codes them):
reg = lm(@formula(log(wage) ~ education + experience + gender + oc), CPS1985)
```

[1] The data set is included in the *R* package **AER**, see https://cran.r-project.org/web/packages/AER/index.html.

```
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")

# rerun regression with different reference category:
reg_newref = lm(@formula(log(wage) ~ education + experience + gender + oc),
    CPS1985,
    contrasts=Dict(:gender => DummyCoding(base="male"),
        :oc => DummyCoding(base="technical")))
table_newref = coeftable(reg_newref)
println("table_newref: \n$table_newref")
```

Output of Script 7.4: `Regr-Categorical.jl`

```
freq_gender:
2-element Named Vector{Int64}
Dim1              |
                  ─────────────────────────────────────────────
String7("female") | 245
String7("male")   | 289

freq_occupation:
6-element Named Vector{Int64}
Dim1                   |
                       ─────────────────────────────────────────────
String15("management") |  55
String15("office")     |  97
String15("sales")      |  38
String15("services")   |  83
String15("technical")  | 105
String15("worker")     | 156

table_reg:

                    Coef.  Std. Error      t  Pr(>|t|)    Lower 95%    Upper 95%

(Intercept)     0.904983    0.171665    5.27    <1e-06   0.567749     1.24222
education       0.0758559   0.0100539   7.54    <1e-12   0.056105     0.0956068
experience      0.0118785   0.0016755   7.09    <1e-11   0.00858694   0.01517
gender: male    0.223848    0.0422525   5.30    <1e-06   0.140843     0.306853
oc: office     -0.207314    0.0776473  -2.67    0.0078  -0.359851    -0.0547764
oc: sales      -0.360111    0.0936446  -3.85    0.0001  -0.544075    -0.176147
oc: services   -0.362586    0.0818392  -4.43    <1e-04  -0.523359    -0.201814
oc: technical  -0.0100857   0.0739841  -0.14    0.8916  -0.155427     0.135255
oc: worker     -0.152544    0.0763434  -2.00    0.0462  -0.30252     -0.00256801

table_newref:

                     Coef.  Std. Error      t  Pr(>|t|)    Lower 95%    Upper 95%

(Intercept)      1.11875     0.176479    6.34    <1e-09   0.772055     1.46544
education        0.0758559   0.0100539   7.54    <1e-12   0.056105     0.0956068
experience       0.0118785   0.0016755   7.09    <1e-11   0.00858694   0.01517
gender: female  -0.223848    0.0422525  -5.30    <1e-06  -0.306853    -0.140843
oc: management   0.0100857   0.0739841   0.14    0.8916  -0.135255     0.155427
oc: office      -0.197228    0.0678171  -2.91    0.0038  -0.330454    -0.064002
oc: sales       -0.350025    0.0863381  -4.05    <1e-04  -0.519636    -0.180415
oc: services    -0.352501    0.0749517  -4.70    <1e-05  -0.499743    -0.205259
oc: worker      -0.142458    0.0704599  -2.02    0.0437  -0.280876    -0.00404035
```

7.4. Breaking a Numeric Variable Into Categories

Sometimes, we do not use a numeric variable directly in a regression model because the implied linear relation seems implausible or inconvenient to interpret. As an alternative to working with transformations such as logs and quadratic terms, it sometimes makes sense to estimate different levels for different ranges of the variable. Wooldridge (2019, Example 7.8) gives the example of the ranking of a law school and how it relates to the starting salary of its graduates.

Given a numeric variable, we need to generate a categorical variable to represent the range into which the rank of a school falls. In *Julia*, the command **`cut`** included in the package **`CategoricalArrays`** is very convenient for this. It takes a numeric variable and a vector of cut points and returns a categorical variable. The lower cut points are included and upper cut points are excluded in the corresponding range.

Wooldridge, Example 7.8: Effects of Law School Rankings on Starting Salaries

The variable **`rank`** of the data set `LAWSCH85` is the rank of the law school as a number between 1 and 175. We would like to compare schools in the top 10, ranks 11–25, 26–40, 41–60, and 61–100 to the reference group of ranks above 100. So in Script 7.5 (`Example-7-8.jl`), we store the cut points 1, 11, 26, 41, 61, 101, and 176 in a variable **`cutpts`**. In the data frame **`lawsch85`**, we create our new variable **`rc`** using the **`cut`** command.
To be consistent with Wooldridge (2019), we do not want the top 10 schools as a reference category but the last category. It is chosen with the **`base`** argument in **`DummyCoding`**. The regression results imply that graduates from the top 10 schools collect a starting salary which is around 70% higher than those of the schools below rank 100. In fact, this approximation is inaccurate with these large numbers and the coefficient of 0.7 actually implies a difference of $\exp(0.7)$-1=1.013 or 101.3%.

Script 7.5: **`Example-7-8.jl`**

```
using WooldridgeDatasets, GLM, DataFrames, CategoricalArrays, FreqTables

lawsch85 = DataFrame(wooldridge("lawsch85"))

# define cut points for the rank:
cutpts = [1, 11, 26, 41, 61, 101, 176]
# note that "cut" takes intervals only in the form of [lower, upper)

# create categorical variable containing ranges for the rank:
lawsch85.rc = cut(lawsch85.rank, cutpts,
        labels=["[1,11)", "[11,26)", "[26,41)",
                "[41,61)", "[61,101)", "[101,176)"])

# display frequencies:
freq = freqtable(lawsch85.rc)
println("freq: \n$freq\n")

# run regression:
reg = lm(@formula(log(salary) ~ rc + LSAT + GPA + log(libvol) + log(cost)),
        lawsch85,
        contrasts=Dict(:rc => DummyCoding(base="[101,176)",
                levels=["[1,11)", "[11,26)", "[26,41)",
                        "[41,61)", "[61,101)", "[101,176)"])))
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Output of Script 7.5: `Example-7-8.jl`

```
freq:
6-element Named Vector{Int64}
Dim1        |
            ---------------------------------------------
"[1,11)"    | 10
"[11,26)"   | 16
"[26,41)"   | 13
"[41,61)"   | 18
"[61,101)"  | 37
"[101,176)" | 62

table_reg:

                    Coef.  Std. Error      t  Pr(>|t|)     Lower 95%  Upper 95%

(Intercept)   9.1653       0.411424    22.28    <1e-44   8.3511        9.97949
rc: [1,11)    0.699566     0.0534919   13.08    <1e-24   0.593707      0.805425
rc: [11,26)   0.593543     0.03944     15.05    <1e-29   0.515493      0.671594
rc: [26,41)   0.375076     0.0340812   11.01    <1e-19   0.307631      0.442522
rc: [41,61)   0.262819     0.0279621    9.40    <1e-15   0.207483      0.318155
rc: [61,101)  0.131595     0.0210419    6.25    <1e-08   0.0899538     0.173236
LSAT          0.00569085   0.00306301   1.86    0.0655  -0.000370751   0.0117524
GPA           0.0137255    0.0741919    0.19    0.8535  -0.133098      0.160549
log(libvol)   0.0363619    0.0260165    1.40    0.1647  -0.015124      0.0878478
log(cost)     0.000841189  0.025136     0.03    0.9734  -0.0489023     0.0505847
```

7.5. Interactions and Differences in Regression Functions Across Groups

Dummy and categorical variables can be interacted just like any other variable. Wooldridge (2019, Section 7.4) discusses the specification and interpretation in this setup. An important case is a model in which one or more dummy variables are interacted with all other regressors. This allows the whole regression model to differ by groups of observations identified by the dummy variable(s).

The example from Wooldridge (2019, Section 7.4-c) is replicated in Script 7.6 (`Dummy-Interact.jl`). Note that the example only applies to the subset of data from **`spring`**. We use the **`subset`** option of **`lm`** directly to define the estimation sample. Other than that, the script does not introduce any new syntax but combines two tricks we have seen previously:

- The dummy variable **`female`** is interacted with all other regressors using the "**`*`**" formula syntax with the other variables contained in parentheses, see Section 6.1.6.
- The *F* test for all interaction effects is performed using the command **`ftest`**.

Script 7.6: Dummy-Interact.jl

```
using WooldridgeDatasets, GLM, DataFrames

gpa3 = DataFrame(wooldridge("gpa3"))

# model with full interactions with female dummy (only for spring data):
reg_ur = lm(@formula(cumgpa ~ female * (sat + hsperc + tothrs)),
    subset(gpa3, :spring => ByRow(==(1))))
table_reg_ur = coeftable(reg_ur)
println("table_reg_ur: \n$table_reg_ur\n")

# F test for H0 (the interaction coefficients of "female" are zero):
reg_r = lm(@formula(cumgpa ~ sat + hsperc + tothrs),
    subset(gpa3, :spring => ByRow(==(1))))

ftest_res = ftest(reg_r.model, reg_ur.model)
fstat = ftest_res.fstat[2]
fpval = ftest_res.pval[2]
println("fstat = $fstat\n")
println("fpval = $fpval")
```

Output of Script 7.6: Dummy-Interact.jl

```
table_reg_ur:
                        Coef.   Std. Error      t  Pr(>|t|)     Lower 95%    Upper 95%

(Intercept)       1.48081      0.207334      7.14    <1e-11   1.07307       1.88856
female           -0.353486     0.410529     -0.86    0.3898  -1.16084       0.453866
sat               0.00105164   0.000181089   5.81    <1e-07   0.000695512   0.00140778
hsperc           -0.00845156   0.00137036   -6.17    <1e-08  -0.0111465    -0.00575659
tothrs            0.00234413   0.000862373   2.72    0.0069   0.000648173   0.00404008
female & sat      0.000750598  0.000385168   1.95    0.0521  -6.87813e-6    0.00150807
female & hsperc  -0.000549755  0.00316172   -0.17    0.8621  -0.00676764    0.00566813
female & tothrs  -0.000115833  0.00162769   -0.07    0.9433  -0.00331687    0.00308521


fstat = 8.179111637046121

fpval = 2.5446371918227897e-6
```

We can estimate the same model parameters by running two separate regressions, one for females and one for males, see Script 7.7 (`Dummy-Interact-Sep.jl`). We see that in the joint model, the parameters without interactions (**`Intercept`**, **`sat`**, **`hsperc`**, and **`tothrs`**) apply to the males and the interaction parameters reflect the *differences* to the males.

To reconstruct the parameters for females from the joint model, we need to add the two respective parameters. The intercept for females is $1.4808 - 0.3535 = 1.1273$ and the coefficient of **`sat`** for females is $0.0011 + 0.0008 \approx 0.0018$.

Script 7.7: `Dummy-Interact-Sep.jl`

```
using WooldridgeDatasets, GLM, DataFrames

gpa3 = DataFrame(wooldridge("gpa3"))

# estimate model for males (& spring data):
reg_m = lm(@formula(cumgpa ~ sat + hsperc + tothrs),
    subset(gpa3, :spring => ByRow(==(1)), :female => ByRow(==(0))))
table_reg_m = coeftable(reg_m)
println("table_reg_m: \n$table_reg_m")

# estimate model for females (& spring data):
reg_f = lm(@formula(cumgpa ~ sat + hsperc + tothrs),
    subset(gpa3, :spring => ByRow(==(1)), :female => ByRow(==(1))))
table_reg_f = coeftable(reg_f)
println("table_reg_f: \n$table_reg_f")
```

Output of Script 7.7: `Dummy-Interact-Sep.jl`

```
table_reg_m:

                  Coef.   Std. Error      t  Pr(>|t|)     Lower 95%    Upper 95%

(Intercept)   1.48081     0.205971       7.19    <1e-11   1.07531       1.88631
sat           0.00105164  0.000179899    5.85    <1e-07   0.000697474   0.00140582
hsperc       -0.00845156  0.00136135    -6.21    <1e-08  -0.0111317    -0.00577144
tothrs        0.00234413  0.000856704    2.74    0.0066   0.000657514   0.00403074

table_reg_f:

                  Coef.   Std. Error      t  Pr(>|t|)     Lower 95%    Upper 95%

(Intercept)   1.12733     0.361595       3.12    0.0025   0.408498      1.84615
sat           0.00180224  0.000346916    5.20    <1e-05   0.0011126     0.00249189
hsperc       -0.00900131  0.00290777    -3.10    0.0027  -0.0147818    -0.00322086
tothrs        0.00222829  0.00140879     1.58    0.1174  -0.000572284   0.00502887
```

8. Heteroscedasticity

The homoscedasticity assumptions SLR.5 for the simple regression model and MLR.5 for the multiple regression model require that the variance of the error terms is unrelated to the regressors, i.e.

$$\text{Var}(u|x_1,\ldots,x_k) = \sigma^2. \tag{8.1}$$

Unbiasedness and consistency (Theorems 3.1, 5.1) do not depend on this assumption, but the sampling distribution (Theorems 3.2, 4.1, 5.2) does. If homoscedasticity is violated, the standard errors are invalid and all inferences from t, F and other tests based on them are unreliable. Also the (asymptotic) efficiency of OLS (Theorems 3.4, 5.3) depends on homoscedasticity. Generally, homoscedasticity is difficult to justify from theory. Different kinds of individuals might have different amounts of unobserved influences in ways that depend on regressors.

We cover three topics: Section 8.1 shows how the formula of the estimated variance-covariance can be adjusted so it does not require homoscedasticity. In this way, we can use OLS to get unbiased and consistent parameter estimates and draw inference from valid standard errors and tests. Section 8.2 presents tests for the existence of heteroscedasticity. Section 8.3 discusses weighted least squares (WLS) as an alternative to OLS. This estimator can be more efficient in the presence of heteroscedasticity.

8.1. Heteroscedasticity-Robust Inference

Wooldridge (2019, Equation 8.4 in Section 8.2) presents the following formula for the classical version of White's heteroscedasticity-robust standard errors:

$$\widehat{Var}(\hat{\beta}_j) = \frac{\sum_{i=1}^{n} \hat{r}_{ij}^2 \cdot \hat{u}_i^2}{(\sum_{i=1}^{n} \hat{r}_{ij}^2)^2} \tag{8.2}$$

In this equation

- $\hat{r}_{ij}$ denotes the i-th residual from regressing x_j on all other explanatory variables, and
- $\hat{u}_i$ is the i-th residual from regressing y on all explanatory variables.

Wooldridge (2010, Formula 4.11) shows another formula for the complete variance-covariance matrix:

$$\widehat{VCov}(\hat{\boldsymbol{\beta}}) = (\mathbf{X}'\mathbf{X})^{-1}(\sum_{i=1}^{n} \hat{u}_i^2 x_i' x_i)(\mathbf{X}'\mathbf{X})^{-1} \tag{8.3}$$

The diagonal of this matrix is equivalent to Equation 8.2. Because we know matrix algebra in *Julia*, we choose this representation to implement heteroscedasticity-robust standard errors in Script 8.1 (`calc-white-se.jl`).[1] The function **`getMats`** was introduced in Section 3.2 in case you want to

[1] We are aware of several packages like **`CovarianceMatrices`** implementing different forms of robust standard errors. However, these packages are either no longer maintained, no longer work in the current *Julia* version, or are documented in a way that beginners may find difficult. Therefore, we have decided against the black box approach.

look it up. As you can see in Scripts 8.2 (`Example-8-2-manual.jl`) and 8.3 (`Example-8-2.jl`), both formulas give the same heteroscedasticity-robust standard errors. Note that t statistics and their p values returned by the **coeftable** function are still based on usual standard errors and need to be updated by Equations 4.6 and 4.7 discussed in Chapter 4.

Script 8.1: **calc-white-se.jl**

```
using LinearAlgebra
include("../03/getMats.jl")

# for details, see Wooldridge (2010), p. 57
function calc_white_se(reg, df)
    f = formula(reg)
    xy = getMats(f, df)
    y = xy[1]
    X = xy[2]
    u = residuals(reg)
    invXX = inv(X' * X)
    sumterm = (X .* u)' * (X .* u)
    avar = invXX' * sumterm * invXX
    std_white = sqrt.(diag(avar))
    return std_white
end
```

Wooldridge, Example 8.2: Heteroscedasticity-Robust Inference

Scripts 8.2 (`Example-8-2-manual.jl`) and 8.3 (`Example-8-2.jl`) demonstrate the calculation of heteroscedasticity-robust standard errors. As you can see, Equation 8.2 and the implementation in Script 8.1 (`calc-white-se.jl`) give the same results. The output of Script 8.3 (`Example-8-2.jl`) also compares to the usual standard errors.
In Script 8.4 (`Example-8-2-cont.jl`) we want to perform an F test and face the problem that there is no implementation available for an F test with heteroscedasticity-robust standard errors. This is one of the few times we use the **PyCall** package to easily access functions that are available in *Python* as introduced in Chapter 1.2.5. Details about the *Python* implementation are given, for example, in Heiss and Brunner (2020). As you can see, the *Python* command **f_test** from the **statsmodels** module is performing the F test and the default of the **fit** method are usual standard errors just as in **GLM**. It is reassuring that the **statsmodels** and **GLM** implementations give the same results. We can also use **cov_type="HC0"** for an F test with heteroscedasticity-robust standard errors.
The results generally do not differ a lot between the different versions. This is an indication that heteroscedasticity might not be a big issue in this example. To be sure, we would like to have a formal test as discussed in the next section.

Script 8.2: **Example-8-2-manual.jl**

```
using WooldridgeDatasets, GLM, DataFrames
include("../03/getMats.jl")
gpa3 = DataFrame(wooldridge("gpa3"))

reg_default = lm(@formula(cumgpa ~ sat + hsperc + tothrs +
                                   female + black + white),
        subset(gpa3, :spring => ByRow(==(1))))
```

```
# extract formula parts for SE calculation:
f = formula(reg_default)
xy = getMats(f, subset(gpa3, :spring => ByRow(==(1))))
y = xy[1]
X = xy[2]
u = residuals(reg_default)
df = DataFrame(X, :auto)


# calculate all rij:
ri1 = residuals(lm(@formula(x1 ~ 0 + x2 + x3 + x4 + x5 + x6 + x7), df))
ri2 = residuals(lm(@formula(x2 ~ 0 + x1 + x3 + x4 + x5 + x6 + x7), df))
ri3 = residuals(lm(@formula(x3 ~ 0 + x1 + x2 + x4 + x5 + x6 + x7), df))
ri4 = residuals(lm(@formula(x4 ~ 0 + x1 + x2 + x3 + x5 + x6 + x7), df))
ri5 = residuals(lm(@formula(x5 ~ 0 + x1 + x2 + x3 + x4 + x6 + x7), df))
ri6 = residuals(lm(@formula(x6 ~ 0 + x1 + x2 + x3 + x4 + x5 + x7), df))
ri7 = residuals(lm(@formula(x7 ~ 0 + x1 + x2 + x3 + x4 + x5 + x6), df))

# calculate SE according to Wooldridge (2019), Ch. 8.2:
se1 = sqrt(sum((ri1 .^ 2) .* (u .^ 2)) / (sum((ri1 .^ 2))^2))
se2 = sqrt(sum((ri2 .^ 2) .* (u .^ 2)) / (sum((ri2 .^ 2))^2))
se3 = sqrt(sum((ri3 .^ 2) .* (u .^ 2)) / (sum((ri3 .^ 2))^2))
se4 = sqrt(sum((ri4 .^ 2) .* (u .^ 2)) / (sum((ri4 .^ 2))^2))
se5 = sqrt(sum((ri5 .^ 2) .* (u .^ 2)) / (sum((ri5 .^ 2))^2))
se6 = sqrt(sum((ri6 .^ 2) .* (u .^ 2)) / (sum((ri6 .^ 2))^2))
se7 = sqrt(sum((ri7 .^ 2) .* (u .^ 2)) / (sum((ri7 .^ 2))^2))

se_white = round.([se1, se2, se3, se4, se5, se6, se7], digits=5)
println("se_white = $se_white")
```

Output of Script 8.2: `Example-8-2-manual.jl`

```
se_white = [0.21856, 0.00019, 0.0014, 0.00073, 0.05857, 0.1181, 0.11032]
```

Script 8.3: `Example-8-2.jl`

```
using WooldridgeDatasets, GLM, DataFrames
include("calc-white-se.jl")

gpa3 = DataFrame(wooldridge("gpa3"))

reg_default = lm(@formula(cumgpa ~ sat + hsperc + tothrs +
                                   female + black + white),
        subset(gpa3, :spring => ByRow(==(1))))

hc0 = calc_white_se(reg_default, subset(gpa3, :spring => ByRow(==(1))))

table_se = DataFrame(coefficients=coeftable(reg_default).rownms,
        b=round.(coef(reg_default), digits=5),
        se_default=round.(coeftable(reg_default).cols[2], digits=5),
        se_white=hc0)
println("table_se: \n$table_se")
```

Output of Script 8.3: Example-8-2.jl

```
table_se:
7×4 DataFrame
 Row | coefficients  b         se_default  se_white
     | String        Float64   Float64     Float64
     ---------------------------------------------
   1 | (Intercept)    1.47006     0.2298   0.21856
   2 | sat            0.00114     0.00018  0.000189691
   3 | hsperc        -0.00857     0.00124  0.0014043
   4 | tothrs         0.0025      0.00073  0.000733526
   5 | female         0.30343     0.05902  0.0585696
   6 | black         -0.12828     0.14737  0.118095
   7 | white         -0.05872     0.14099  0.110322
```

Script 8.4: Example-8-2-cont.jl

```
using PyCall, WooldridgeDatasets, GLM, DataFrames
include("../03/getMats.jl")

# install Python's statsmodels with: using Conda; Conda.add("statsmodels")
sm = pyimport("statsmodels.api")

gpa3 = DataFrame(wooldridge("gpa3"))
gpa3_subset = subset(gpa3, :spring => ByRow(==(1)))

# F test using usual VCOV in Julia:
reg_ur = lm(@formula(cumgpa ~ sat + hsperc + tothrs + female + black + white),
    gpa3_subset)
reg_r = lm(@formula(cumgpa ~ sat + hsperc + tothrs + female),
    gpa3_subset)
ftest_res = ftest(reg_r.model, reg_ur.model)
fstat_jl = ftest_res.fstat[2]
fpval_jl = ftest_res.pval[2]
println("fstat_jl = $fstat_jl\n")
println("fpval_jl = $fpval_jl\n")

# F test using different variance-covariance formulas:
# definition of model and hypotheses:
f = @formula(cumgpa ~ 1 + sat + hsperc + tothrs + female + black + white)
xy = getMats(f, gpa3_subset)
reg = sm.OLS(xy[1], xy[2])
hypotheses = ["x5 = 0", "x6 = 0"] # meaning "black = 0" and "white = 0"

# usual VCOV in Python:
results_default = reg.fit()
ftest_py_default = results_default.f_test(hypotheses)
fstat_py_default = ftest_py_default.statistic
fpval_py_default = ftest_py_default.pvalue
println("fstat_py_default = $fstat_py_default\n")
println("fpval_py_default = $fpval_py_default\n")

# classical White VCOV in Python:
results_hc0 = reg.fit(cov_type="HC0")
ftest_py_hc0 = results_hc0.f_test(hypotheses)
fstat_py_hc0 = ftest_py_hc0.statistic
fpval_py_hc0 = ftest_py_hc0.pvalue
println("fstat_py_hc0 = $fstat_py_hc0\n")
println("fpval_py_hc0 = $fpval_py_hc0")
```

Output of Script 8.4: `Example-8-2-cont.jl`

```
fstat_jl = 0.6796041956073717

fpval_jl = 0.507468362258422

fstat_py_default = 0.6796041956073425

fpval_py_default = 0.5074683622584049

fstat_py_hc0 = 0.7477969818036305

fpval_py_hc0 = 0.4741442714738484
```

8.2. Heteroscedasticity Tests

The Breusch-Pagan (BP) test for heteroscedasticity is easy to implement with basic OLS routines. After a model

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k + u \tag{8.4}$$

is estimated, we obtain the residuals $\hat{u}_i$ for all observations $i = 1, \ldots, n$. We regress their squared value on all independent variables from the original equation. We can either perform the standard F test of overall significance with **`ftest`**. Or we can use an LM test by multiplying the R^2 from the second regression with the number of observations.

In **`GLM`**, this is easily done. Remember that the residuals from a regression can be obtained by the **`residuals`** function. Their squared value can be stored in a new variable to be used as a dependent variable in the second stage.

The LM version of the BP test is even more convenient to use with the function **`WhiteTest`** from the **`HypothesisTests`** package. The computation of the test statistic and corresponding p value is demonstrated in Script 8.5 (`Example-8-4.jl`).

Wooldridge, Example 8.4: Heteroscedasticity in a Housing Price Equation

Script 8.5 (`Example-8-4.jl`) implements the F and LM versions of the BP test. The command **`WhiteTest`** simply takes the regressor matrix, the regression residuals, and **`type=:linear`** for the Breusch-Pagan test as arguments and delivers a test statistic of $LM = 14.09$. The corresponding p value is smaller than 0.003 so we reject homoscedasticity for all reasonable significance levels.
The output also shows the manual implementation of a second stage regression where we regress squared residuals on the independent variables. We can directly interpret the reported F statistic of 5.34 and its p value of 0.002 as the F version of the BP test. We can manually calculate the LM statistic by multiplying R^2 $(= 0.16)$ with the number of observations $(n = 88)$.
We replicate the test for an alternative model with logarithms discussed by Wooldridge (2019) together with the White test in Example 8.5 and Script 8.6 (`Example-8-5.jl`).

Script 8.5: `Example-8-4.jl`

```
using WooldridgeDatasets, GLM, DataFrames, HypothesisTests

hprice1 = DataFrame(wooldridge("hprice1"))

# estimate model:
f = @formula(price ~ 1 + lotsize + sqrft + bdrms)
reg = lm(f, hprice1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg\n")

# automatic BP test (LM version),
# type = :linear specifies Breusch-Pagan test:
X = getMats(f, hprice1)[2]
result_bp_lm = WhiteTest(X, residuals(reg), type=:linear)
bp_lm_statistic = result_bp_lm.lm
bp_lm_pval = pvalue(result_bp_lm)
println("bp_lm_statistic = $bp_lm_statistic\n")
println("bp_lm_pval = $bp_lm_pval\n")

# manual BP test (F version):
hprice1.resid_sq = residuals(reg) .^ 2
reg_resid = lm(@formula(resid_sq ~ lotsize + sqrft + bdrms), hprice1)
bp_F = ftest(reg_resid.model)
bp_F_statistic = bp_F.fstat
bp_F_pval = bp_F.pval
println("bp_F_statistic = $bp_F_statistic\n")
println("bp_F_pval = $bp_F_pval")
```

Output of Script 8.5: `Example-8-4.jl`

```
table_reg:

                   Coef.    Std. Error      t  Pr(>|t|)      Lower 95%    Upper 95%

(Intercept)  -21.7703      29.475         -0.74    0.4622  -80.3847       36.844
lotsize        0.00206771   0.000642126   3.22    0.0018    0.000790769   0.00334464
sqrft          0.122778     0.0132374     9.28    <1e-13    0.0964541     0.149102
bdrms         13.8525       9.01015       1.54    0.1279   -4.06514      31.7702


bp_lm_statistic = 14.092385504350007

bp_lm_pval = 0.0027820595556893894

bp_F_statistic = 5.338919363241315

bp_F_pval = 0.002047744420936324
```

The White test is a variant of the BP test where in the second stage, we do not regress the squared first-stage residuals on the original regressors only. Instead, we add interactions and polynomials of them or include the fitted values $\hat{y}$ and $\hat{y}^2$. This can easily be done in a manual second-stage regression remembering that the fitted values are computed by **`predict`**.

Conveniently, we can also use the **`WhiteTest`** command to do the calculations of the *LM* version of the test including the p values automatically. All we have to do is to explain that in the second stage we want a different set of regressors.

Wooldridge, Example 8.5: BP and White test in the Log Housing Price Equation

Script 8.6 (Example-8-5.jl) implements the BP and the White test for a model that now contains logarithms of the dependent variable and three independent variables. The LM versions of both the BP and the White test do not reject the null hypothesis at conventional significance levels with p values of 0.238 and 0.178, respectively.

Script 8.6: Example-8-5.jl

```
using WooldridgeDatasets, GLM, DataFrames, HypothesisTests
include("../03/getMats.jl")

hprice1 = DataFrame(wooldridge("hprice1"))

# estimate model:
f = @formula(log(price) ~ 1 + log(lotsize) + log(sqrft) + bdrms)
reg = lm(f, hprice1)

# BP test:
X = getMats(f, hprice1)[2]
result_bp = WhiteTest(X, residuals(reg), type=:linear)
bp_statistic = result_bp.lm
bp_pval = pvalue(result_bp)
println("bp_statistic = $bp_statistic\n")
println("bp_pval = $bp_pval\n")

# White test:
X_wh = hcat(ones(size(X)[1]),
    predict(reg),
    predict(reg) .^ 2)
result_white = WhiteTest(X_wh, residuals(reg), type=:linear)
white_statistic = result_white.lm
white_pval = pvalue(result_white)
println("white_statistic = $white_statistic\n")
println("white_pval = $white_pval")
```

Output of Script 8.6: Example-8-5.jl

```
bp_statistic = 4.2232457418006355

bp_pval = 0.2383448263153909

white_statistic = 3.4472865468700427

white_pval = 0.17841494794177618
```

8.3. Weighted Least Squares

Weighted Least Squares (WLS) attempts to provide a more efficient alternative to OLS. It is a special version of a feasible generalized least squares (FGLS) estimator. Instead of the sum of squared residuals, their weighted sum is minimized. If the weights are inversely proportional to the variance, the estimator is efficient. Also the usual formula for the variance-covariance matrix of the parameter estimates and standard inference tools are valid.

We can obtain WLS parameter estimates by multiplying each variable in the model with the square root of the weight as shown by Wooldridge (2019, Section 8.4) and demonstrated in Script 8.7 (`Example-8-6.jl`).

Wooldridge, Example 8.6: Financial Wealth Equation

Script 8.7 (`Example-8-6.jl`) implements both OLS and WLS estimation for a regression of financial wealth (**`nettfa`**) on income (**`inc`**), age (**`age`**), gender (**`male`**) and eligibility for a pension plan (**`e401k`**) using the data set `401ksubs`. Following Wooldridge (2019), we assume that the variance is proportional to the income variable **`inc`**. Therefore, the optimal weight is $\frac{1}{\texttt{inc}}$, which is considered in the **`identity`** calls within the formula. The weighting of the constant is implemented by including the weight itself in the regression model ($1 \cdot \frac{1}{\sqrt{inc}} =$ **`w`** in the formula) and excluding the original constant (**`0`** in the formula).

Script 8.7: `Example-8-6.jl`

```
using WooldridgeDatasets, GLM, DataFrames
include("calc-white-se.jl")

k401ksubs = DataFrame(wooldridge("401ksubs"))

# subsetting data:
k401ksubs_sub = subset(k401ksubs, :fsize => ByRow(==(1)))

# OLS (only for singles, i.e. 'fsize'==1):
reg_ols = lm(@formula(nettfa ~ inc + ((age - 25)^2) + male + e401k),
        k401ksubs_sub)
hc0 = calc_white_se(reg_ols, k401ksubs_sub)

# print regression table with hc0:
table_ols = DataFrame(coefficients=coeftable(reg_ols).rownms,
        b=round.(coef(reg_ols), digits=5),
        se=round.(hc0, digits=5))
println("table_ols: \n$table_ols\n")

# WLS:
k401ksubs_sub.w = (1 ./ sqrt.(k401ksubs_sub.inc))
reg_wls = lm(@formula(identity(nettfa * w) ~ 0 + w + identity(inc * w) +
                                             identity((age - 25)^2 * w) +
                                             identity(male * w) +
                                             identity(e401k * w)), k401ksubs_sub)

# print regression table:
table_wls = DataFrame(coefficients=coeftable(reg_wls).rownms,
        b=round.(coef(reg_wls), digits=5),
        se=round.(coeftable(reg_wls).cols[2], digits=5))
println("table_wls: \n$table_wls\n")
```

Output of Script 8.7: `Example-8-6.jl`

```
table_ols:
5×3 DataFrame
 Row | coefficients    b          se
     | String          Float64    Float64
     --------------------------------------------
   1 | (Intercept)     -20.985    3.49085
   2 | inc               0.77058  0.09945
   3 | (age - 25) ^ 2    0.02513  0.00434
   4 | male              2.47793  2.05581
   5 | e401k             6.88622  2.28374

table_wls:
5×3 DataFrame
 Row | coefficients                b          se
     | Any                         Float64    Float64
     --------------------------------------------
   1 | w                           -16.7025   1.95799
   2 | identity(inc * w)             0.74038  0.0643
   3 | identity((age - 25) ^ 2 * w)  0.01754  0.00193
   4 | identity(male * w)            1.84053  1.56359
   5 | identity(e401k * w)           5.18828  1.70343
```

We can also use heteroscedasticity-robust statistics from Section 8.1 to account for the fact that our variance function might be misspecified. Script 8.8 (`WLS-Robust.jl`) repeats the WLS estimation of Example 8.6 but reports non-robust and robust standard errors. It replicates Wooldridge (2019, Table 8.2) and there is nothing special about the implementation. The fact that we used weights is correctly accounted for in the following calculations.

Script 8.8: `WLS-Robust.jl`

```
using WooldridgeDatasets, GLM, DataFrames
include("calc-white-se.jl")

k401ksubs = DataFrame(wooldridge("401ksubs"))

# subsetting data:
k401ksubs_sub = subset(k401ksubs, :fsize => ByRow(==(1)))

# WLS:
k401ksubs_sub.w = (1 ./ sqrt.(k401ksubs_sub.inc))
reg_wls = lm(@formula(identity(nettfa * w) ~ 0 + w + identity(inc * w) +
                                             identity((age - 25)^2 * w) +
                                             identity(male * w) +
                                             identity(e401k * w)), k401ksubs_sub)

# robust results (White SE):
hc0 = calc_white_se(reg_wls, k401ksubs_sub)

# print regression table:
table_default = DataFrame(coefficients=coeftable(reg_wls).rownms,
        b=round.(coef(reg_wls), digits=5),
        se_default=round.(coeftable(reg_wls).cols[2], digits=5),
        se_robust=round.(hc0, digits=5))
println("table_default: \n$table_default")
```

Output of Script 8.8: `WLS-Robust.jl`

```
table_default:
5×4 DataFrame
 Row | coefficients                     b           se_default  se_robust
     | Any                              Float64     Float64     Float64
     ---------------------------------------------
   1 | w                               -16.7025      1.95799    2.2402
   2 | identity(inc * w)                 0.74038     0.0643     0.07496
   3 | identity((age - 25) ^ 2 * w)      0.01754     0.00193    0.00258
   4 | identity(male * w)                1.84053     1.56359    1.30918
   5 | identity(e401k * w)               5.18828     1.70343    1.56991
```

The assumption made in Example 8.6 that the variance is proportional to a regressor is usually hard to justify. Typically, we don't know the variance function and have to estimate it. This feasible GLS (FGLS) estimator replaces the (allegedly) known variance function with an estimated one.

We can estimate the relation between variance and regressors using a linear regression of the log of the squared residuals from an initial OLS regression $\log(\hat{u}^2)$ as the dependent variable. Wooldridge (2019, Section 8.4) suggests two versions for the selection of regressors:

- the regressors $x_1, \ldots, x_k$ from the original model similar to the BP test
- $\hat{y}$ and $\hat{y}^2$ from the original model similar to the White test

As the estimated error variance, we can use $\exp\left(\widehat{\log(\hat{u}^2)}\right)$. Its inverse can then be used as a weight in WLS estimation.

Wooldridge, Example 8.7: Demand for Cigarettes

Script 8.9 (`Example-8-7.jl`) studies the relationship between daily cigarette consumption **`cigs`**, individual characteristics, and restaurant smoking restrictions **`restaurn`**. After the initial OLS regression, a BP test is performed which clearly rejects homoscedasticity (see previous section for the BP test). After the regression of log squared residuals on the regressors, the FGLS weights are calculated and used in the WLS regression. See Wooldridge (2019) for a discussion of the results.

Script 8.9: `Example-8-7.jl`

```
using WooldridgeDatasets, GLM, DataFrames, HypothesisTests

smoke = DataFrame(wooldridge("smoke"))

# OLS:
reg_ols = lm(@formula(cigs ~ log(income) + log(cigpric) +
                             educ + age + age^2 + restaurn), smoke)
table_ols = DataFrame(coefficients=coeftable(reg_ols).rownms,
        b=round.(coef(reg_ols), digits=5),
        se=round.(stderror(reg_ols), digits=5))
println("table_ols: \n$table_ols\n")

# BP test:
X = modelmatrix(reg_ols)
result_bp = WhiteTest(X, residuals(reg_ols), type=:linear)
bp_statistic = result_bp.lm
bp_pval = pvalue(result_bp)
println("bp_statistic = $bp_statistic\n")
println("bp_pval = $bp_pval\n")

# FGLS (estimation of the variance function):
smoke.logu2 = log.(residuals(reg_ols) .^ 2)
reg_fgls = lm(@formula(logu2 ~ log(income) + log(cigpric) +
                               educ + age + age^2 + restaurn), smoke)

table_fgls = DataFrame(coefficients=coeftable(reg_fgls).rownms,
        b=round.(coef(reg_fgls), digits=5),
        se=round.(stderror(reg_fgls), digits=5))
println("table_fgls: \n$table_fgls\n")

# FGLS (WLS):
smoke.w = (1 ./ sqrt.(exp.(predict(reg_fgls))))

reg_wls = lm(@formula(identity(cigs * w) ~ 0 + w + identity(log(income) * w) +
                                           identity(log(cigpric) * w) +
                                           identity(educ * w) +
                                           identity(age * w) +
                                           identity(age^2 * w) +
                                           identity(restaurn * w)), smoke)

table_wls = DataFrame(coefficients=coeftable(reg_wls).rownms,
        b=round.(coef(reg_wls), digits=5),
        se=round.(stderror(reg_wls), digits=5))
println("table_wls: \n$table_wls")
```

Output of Script 8.9: `Example-8-7.jl`

```
table_ols:
7×3 DataFrame
 Row | coefficients  b         se
     | String        Float64   Float64
     ---------------------------------------------
   1 | (Intercept)   -3.63983  24.0787
   2 | log(income)    0.88027   0.72778
   3 | log(cigpric)  -0.75086   5.77334
   4 | educ          -0.5015    0.16708
   5 | age            0.77069   0.16012
   6 | age ^ 2       -0.00902   0.00174
   7 | restaurn      -2.82508   1.11179

bp_statistic = 32.25841908120354

bp_pval = 1.4557794830263948e-5

table_fgls:
7×3 DataFrame
 Row | coefficients  b         se
     | String        Float64   Float64
     ---------------------------------------------
   1 | (Intercept)   -1.92069  2.56303
   2 | log(income)    0.29154  0.07747
   3 | log(cigpric)   0.19542  0.61454
   4 | educ          -0.0797   0.01778
   5 | age            0.20401  0.01704
   6 | age ^ 2       -0.00239  0.00019
   7 | restaurn      -0.62701  0.11834

table_wls:
7×3 DataFrame
 Row | coefficients                  b         se
     | Any                           Float64   Float64
     ---------------------------------------------
   1 | w                              5.63546  17.8031
   2 | identity(log(income) * w)      1.29524   0.43701
   3 | identity(log(cigpric) * w)    -2.94031   4.46014
   4 | identity(educ * w)            -0.46345   0.12016
   5 | identity(age * w)              0.48195   0.09681
   6 | identity(age ^ 2 * w)         -0.00563   0.00094
   7 | identity(restaurn * w)        -3.46106   0.79551
```

9. More on Specification and Data Issues

This chapter covers different topics of model specification and data problems. Section 9.1 asks how statistical tests can help us specify the "correct" functional form given the numerous options we have seen in Chapters 6 and 7. Section 9.2 shows some simulation results regarding the effects of measurement errors in dependent and independent variables. Sections 9.3 covers missing values and how *Julia* can deal with them. In Section 9.4, we briefly discuss outliers and Section 9.5, the LAD estimator is presented.

9.1. Functional Form Misspecification

We have seen many ways to flexibly specify the relation between the dependent variable and the regressors. An obvious question to ask is whether or not a given specification is the "correct" one. The Regression Equation Specification Error Test (RESET) is a convenient tool to test the null hypothesis that the functional form is adequate.

Wooldridge (2019, Section 9.1) shows how to implement it using a standard F test in a second regression that contains polynomials of fitted values from the original regression. We already know how to obtain fitted values and run an F test, so the implementation is straightforward.

Wooldridge, Example 9.2: Housing Price Equation

Script 9.1 (`Example-9-2.jl`) implements the RESET test using the procedure described by Wooldridge (2019) for the housing price model. As previously, we get the fitted values from the original regression using **`predict`**. Their polynomials are entered into the formula of the second regression. To avoid numerical problems, predictions are divided by 1000. The F test is easily done using **`ftest`** as described in Section 4.3. The test statistic is $F = 4.67$ with a p value of $p = 0.012$, so we reject the null hypothesis that this equation is correctly specified at a significance level of $\alpha = 5\%$.

Script 9.1: `Example-9-2.jl`

```
using WooldridgeDatasets, GLM, DataFrames

hprice1 = DataFrame(wooldridge("hprice1"))

# original OLS:
reg = lm(@formula(price ~ lotsize + sqrft + bdrms), hprice1)

# regression for RESET test:
hprice1.fitted_sq = predict(reg) .^ 2 ./ 1000
hprice1.fitted_cub = predict(reg) .^ 3 ./ 1000

reg_reset = lm(@formula(price ~ lotsize + sqrft + bdrms +
                                fitted_sq + fitted_cub), hprice1)
table_reg_reset = coeftable(reg_reset)
println("table_reg_reset: \n$table_reg_reset\n")
```

```
# RESET test (H0: all coefficients including "fitted" are zero):
ftest_res = ftest(reg.model, reg_reset.model)
fstat = ftest_res.fstat[2]
fpval = ftest_res.pval[2]
println("fstat = $fstat\n")
println("fpval = $fpval")
```

Output of Script 9.1: `Example-9-2.jl`

```
table_reg_reset:

                      Coef.    Std. Error     t  Pr(>|t|)     Lower 95%    Upper 95%

(Intercept)  166.097        317.433        0.52    0.6022  -465.377       797.572
lotsize        0.000153723    0.00520304  0.03    0.9765    -0.0101968     0.0105042
sqrft          0.0175989      0.299251    0.06    0.9532    -0.577706      0.612904
bdrms          2.1749        33.8881      0.06    0.9490   -65.2393       69.5891
fitted_sq      0.353426       7.09894     0.05    0.9604   -13.7686       14.4755
fitted_cub     0.00154557     0.00655431  0.24    0.8142    -0.011493      0.0145842


fstat = 4.668205534949378 5

fpval = 0.012021711442876 59
```

Wooldridge (2019, Section 9.1-b) also discusses tests of non-nested models. As an example, a test of both models against a comprehensive model containing all regressors is mentioned. Such a test can be implemented in **GLM** by the command **ftest** that we already discussed. Script 9.2 (`Nonnested-Test.jl`) shows this test in action for a modified version of Example 9.2.

The two alternative models for the housing price are

$$\texttt{price} = \beta_0 + \beta_1 \texttt{lotsize} + \beta_2 \texttt{sqrft} + \beta_3 \texttt{bdrms} + u, \tag{9.1}$$

$$\texttt{price} = \beta_0 + \beta_1 \log(\texttt{lotsize}) + \beta_2 \log(\texttt{sqrft}) + \beta_3 \texttt{bdrms} + u. \tag{9.2}$$

The output shows the test results of testing both models against the encompassing model with all variables. Both models are rejected against this comprehensive model.

Script 9.2: `Nonnested-Test.jl`

```
using WooldridgeDatasets, GLM, DataFrames

hprice1 = DataFrame(wooldridge("hprice1"))

# two alternative models:
reg1 = lm(@formula(price ~ lotsize + sqrft + bdrms), hprice1)
reg2 = lm(@formula(price ~ log(lotsize) + log(sqrft) + bdrms), hprice1)

# encompassing test of Davidson & MacKinnon:
# comprehensive model:
reg3 = lm(@formula(price ~ lotsize + sqrft + bdrms +
                           log(lotsize) + log(sqrft)), hprice1)

# test model 1:
ftest_res1 = ftest(reg1.model, reg3.model)

fstat1 = ftest_res1.fstat[2]
fpval1 = ftest_res1.pval[2]
println("fstat1 = $fstat1\n")
println("fpval1 = $fpval1\n")
```

```
# test model 2:
ftest_res2 = ftest(reg2.model, reg3.model)

fstat2 = ftest_res2.fstat[2]
fpval2 = ftest_res2.pval[2]
println("fstat2 = $fstat2\n")
println("fpval2 = $fpval2")
```

Output of Script 9.2: Nonnested-Test.jl

```
fstat1 = 7.8612911192328445

fpval1 = 0.0007526198013482557

fstat2 = 7.050759706344788

fpval2 = 0.001494264670031476
```

9.2. Measurement Error

If a variable is not measured accurately, the consequences depend on whether the measurement error affects the dependent or an explanatory variable. If the **dependent variable** is mismeasured, the consequences can be mild. If the measurement error is unrelated to the regressors, the parameter estimates get less precise, but they are still consistent and the usual inferences from the results are valid.

The simulation exercise in Script 9.3 (`Sim-ME-Dep.jl`) draws 10,000 samples of size $n = 1,000$ according to the model with measurement error in the dependent variable

$$y^* = \beta_0 + \beta_1 x + u, \qquad y = y^* + e_0. \tag{9.3}$$

The assumption is that we do not observe the true values of the dependent variable y^* but our measure y is contaminated with a measurement error e_0.

Script 9.3: Sim-ME-Dep.jl

```
using Random, Distributions, Statistics, GLM, DataFrames

# set the random seed:
Random.seed!(12345)

# set sample size and number of simulations:
n = 1000
r = 10000

# set true parameters (betas):
beta0 = 1
beta1 = 0.5

# initialize arrays to store results later (b1 without ME, b1_me with ME):
b1 = zeros(r)
b1_me = zeros(r)

# draw a sample of x, fixed over replications:
x = rand(Normal(4, 1), n)
```

```
# repeat r times:
for i in 1:r
        # draw a sample of u:
        u = rand(Normal(0, 1), n)

        # draw a sample of ystar:
        ystar = beta0 .+ beta1 * x .+ u

        # measurement error and mismeasured y:
        e0 = rand(Normal(0, 1), n)
        y = ystar .+ e0
        df = DataFrame(ystar=ystar, y=y, x=x)

        # regress ystar on x and store slope estimate at position i:
        reg_star = lm(@formula(ystar ~ x), df)
        b1[i] = coef(reg_star)[2]

        # regress y on x and store slope estimate at position i:
        reg_me = lm(@formula(y ~ x), df)
        b1_me[i] = coef(reg_me)[2]
end

# mean with and without ME:
b1_mean = mean(b1)
b1_me_mean = mean(b1_me)
println("b1_mean = $b1_mean\n")
println("b1_me_mean = $b1_me_mean\n")

# variance with and without ME:
b1_var = var(b1)
b1_me_var = var(b1_me)
println("b1_var = $b1_var\n")
println("b1_me_var = $b1_me_var")
```

Output of Script 9.3: `Sim-ME-Dep.jl`

```
b1_mean = 0.4998083385316263

b1_me_mean = 0.49971475891456885

b1_var = 0.0009211085272951774

b1_me_var = 0.0018723380569089674
```

In the simulation, the parameter estimates using both the correct y^* and the mismeasured y are stored as the variables **b1** and **b1_me**, respectively. As expected, the simulated mean of both variables is close to the expected value of $\beta_1 = 0.5$. Output 9.3 (`Sim-ME-Dep.jl`) shows that the variance of **b1_me** is around 0.002 which is twice as high as the variance of **b1**. This was expected since in our simulation, u and e_0 are both independent standard normal variables, so $\text{Var}(u) = 1$ and $\text{Var}(u + e_0) = 2$.

If an **explanatory variable** is mismeasured, the consequences are usually more dramatic. Even in the classical errors-in-variables case where the measurement error is unrelated to the regressors, the parameter estimates are biased and inconsistent. This model is

$$y = \beta_0 + \beta_1 x^* + u, \qquad x = x^* + e_1 \tag{9.4}$$

where the measurement error e_1 is independent of both x^* and u. Wooldridge (2019, Section 9.4) shows that if we regress y on x instead of x^*,

$$\text{plim}\hat{\beta}_1 = \beta_1 \cdot \frac{\text{Var}(x^*)}{\text{Var}(x^*) + \text{Var}(e_1)}. \tag{9.5}$$

The simulation in Script 9.4 (`Sim-ME-Explan.jl`) draws 10,000 samples of size $n = 1,000$ from this model.

Script 9.4: `Sim-ME-Explan.jl`

```
using Random, Distributions, Statistics, GLM, DataFrames

# set the random seed:
Random.seed!(12345)

# set sample size and number of simulations:
n = 1000
r = 10000

# set true parameters (betas):
beta0 = 1
beta1 = 0.5

# initialize arrays to store results later (b1 without ME, b1_me with ME):
b1 = zeros(r)
b1_me = zeros(r)

# draw a sample of x, fixed over replications:
xstar = rand(Normal(4, 1), n)

# repeat r times:
for i in 1:r
        # draw a sample of u:
        u = rand(Normal(0, 1), n)

        # draw a sample of y:
        y = beta0 .+ beta1 * xstar .+ u

        # measurement error and mismeasured x:
        e1 = rand(Normal(0, 1), n)
        x = xstar .+ e1
        df = DataFrame(y=y, xstar=xstar, x=x)

        # regress y on xstar and store slope estimate at position i:
        reg_star = lm(@formula(y ~ xstar), df)
        b1[i] = coef(reg_star)[2]

        # regress y on x and store slope estimate at position i:
        reg_me = lm(@formula(y ~ x), df)
        b1_me[i] = coef(reg_me)[2]
end

# mean with and without ME:
b1_mean = mean(b1)
b1_me_mean = mean(b1_me)
println("b1_mean = $b1_mean\n")
println("b1_me_mean = $b1_me_mean\n")
```

```
# variance with and without ME:
b1_var = var(b1)
b1_me_var = var(b1_me)
println("b1_var = $b1_var\n")
println("b1_me_var = $b1_me_var")
```

Output of Script 9.4: `Sim-ME-Explan.jl`

```
b1_mean = 0.4998083385316263

b1_me_mean = 0.2586050939287857

b1_var = 0.0009211085272951774

b1_me_var = 0.0005148989427600099
```

Since in this simulation, $\mathrm{Var}(x^*) = \mathrm{Var}(e_1) = 1$, Equation 9.5 implies that $\mathrm{plim}\hat{\beta}_1 = \frac{1}{2}\beta_1 = 0.25$. This is confirmed by the simulation results in Output 9.4 (`Sim-ME-Explan.jl`). While the mean of the estimates in **`b1`** using the correct regressor again is around 0.5, the mean parameter estimate using the mismeasured regressor is about 0.25.

9.3. Missing Data and Nonrandom Samples

In many data sets, we fail to observe all variables for each observational unit. An important case is survey data where the respondents refuse or fail to answer some questions. We can account for missing data in *Julia* by using the special value **`missing`**. For missing *numeric* values, like the result of a mathematically undefined operation, there is another special value: **`NaN`** (not a number). Both indicate that we do not have the information or the value is not defined.[1]

The function **`ismissing(value)`** returns **`true`** if **`value`** is **`missing`** and **`false`** otherwise. The same applies to the function **`isnan(value)`** if **`value`** is or is not **`NaN`**. The function **`skipmissing`** is useful to exclude missings from further analysis. Operations resulting in $\pm\infty$ like $\log(0)$ or $\frac{1}{0}$ are not coded as **`NaN`** but as **`Inf`** or **`-Inf`**. Also note that mathematically undefined operations can result in an **`ERROR`** or **`NaN`**, but we can always store such a result as **`NaN`** with a **`try`** statement. Script 9.5 (`NaN-Inf-Missing.jl`) gives some examples.

[1]A very illustrative comparison of these particular values can be found here: `https://miguelraz.github.io/blog/nothingforbeginners/index.html`.

Script 9.5: NaN-Inf-Missing.jl

```
using Distributions, DataFrames, Statistics

# NaN, missings and infinite values in Julia:
x1 = [0, 2, NaN, Inf, missing]
logx = log.(x1)
invx = 0 ./ x1
isnanx = isnan.(x1)
isinfx = isinf.(x1)
ismissingx = ismissing.(x1)

results = DataFrame(x1=x1, logx=logx, invx=invx, ismissingx=ismissingx,
    isnanx=isnanx, isinfx=isinfx)
println("results = $results\n")

# mathematically not defined is not always NaN (like in R or Python):
test = try
    log(-1) # results in an ERROR
catch e
    NaN
end
println("test = $test\n")

# handling missings:
x2 = [4, 2, missing, 3]
meanx2_1 = mean(x2)
println("meanx2_1 = $meanx2_1\n")

meanx2_2 = mean(skipmissing(x2))
println("meanx2_2 = $meanx2_2\n")

x3 = [4, 2, NaN, 3]
meanx3_1 = mean(x3)
println("meanx3_1 = $meanx3_1\n")

meanx3_2 = mean(x3[.!isnan.(x3)])
println("meanx3_2 = $meanx3_2")
```

Output of Script 9.5: NaN-Inf-Missing.jl

```
results = 5×6 DataFrame
 Row | x1         logx            invx       ismissingx  isnanx   isinfx
     | Float64?   Float64?        Float64?   Bool        Bool?    Bool?
     ---------------------------------------------
   1 |       0.0     -Inf             NaN        false    false    false
   2 |       2.0        0.693147        0.0      false    false    false
   3 |     NaN        NaN             NaN        false     true    false
   4 |     Inf        Inf               0.0      false    false     true
   5 | missing    missing         missing         true  missing  missing

test = NaN

meanx2_1 = missing

meanx2_2 = 3.0

meanx3_1 = NaN

meanx3_2 = 3.0
```

Depending on the data source, real-world data sets can have different rules for indicating missing information. Sometimes, impossible numeric values are used. For example, a survey including the number of years of education as a variable **educ** might have a value like "9999" to indicate missing information. For any software package, it is highly recommended to change these to proper missing-value codes early in the data-handling process. Otherwise, we take the risk that some statistical method interprets those values as "this person went to school for 9999 years" producing highly nonsensical results. For the education example, if the variable **educ** is in the data frame **mydata** this can be done with either of the two lines of code:

```
mydata.educ = ifelse.(mydata.educ .== 9999, missing, mydata.educ)
mydata.educ = ifelse.(mydata.educ .== 9999, NaN, mydata.educ)
```

If missing values are coded as **missing**, the function **ismissing** can also be applied to the whole data frame. The output is a data frame with the same dimensions and variable names but full of Boolean variables set as **true** for missing observations. If we are interested in the missings for each variable, we can use:

```
miss_all = ismissing.(mydata)
freq_missLSAT = mapcols(count, miss_all)
```

The function **mapcols** applies the **count** function to each column, i.e. variable, in **miss_all**. In the latter, each observation is **true** (treated as **1** by **count**) or **false** (treated as **0** by **count**), which gives the total amount of missing values per variable.

If we are interested in observations, where no variable has a missing value, we can exclude observations with missings by **dropmissing**. As an alternative, you could also use **completecases** which gives a vector of **true** (observation has no missings) and **false** (observation has at least one missing) values. The total amount of complete observations is then simply computed as:

```
mydata_compl_cases = dropmissing(mydata)
complete_cases1 = nrow(mydata_compl_cases)

compl_cases = completecases(mydata)
complete_cases2 = count(compl_cases)
```

Script 9.6 (`Missings.jl`) demonstrates these commands for the data set `LAWSCH85` which contains data on law schools. Of the 156 schools, 6 do not report median LSAT scores. Looking at all variables, the most missings are found for the **age** of the school – we don't know it for 45 schools. For only 90 of the 156 schools, we have the full set of variables, for the other 66, one or more variable is missing.

Script 9.6: `Missings.jl`

```
using WooldridgeDatasets, GLM, DataFrames

lawsch85 = DataFrame(wooldridge("lawsch85"))
lsat = lawsch85.LSAT

# create boolean indicator for missings:
missLSAT = ismissing.(lsat)

# LSAT and indicator for Schools No. 120-129:
preview = DataFrame(lsat=lsat[120:129],
    missLSAT=missLSAT[120:129])
println("preview: \n$preview\n")
```

```
# frequencies of indicator:
tot_missing = count(missLSAT) # same as sum(missLSAT)
tot_nonmissings = count(.!missLSAT)
println("tot_missing = $tot_missing\n")
println("tot_nonmissings = $tot_nonmissings\n")

# missings for all variables in data frame (counts):
miss_all = ismissing.(lawsch85)
freq_missLSAT = mapcols(count, miss_all)
freq_missLSAT_preview = freq_missLSAT[:, 1:9] # print only first nine columns
println("freq_missLSAT_preview: \n$freq_missLSAT_preview\n")

# computing amount of complete cases:
lsat_compl_cases1 = dropmissing(lawsch85)
complete_cases1 = nrow(lsat_compl_cases1)
println("complete_cases1 = $complete_cases1\n")

lsat_compl_cases2 = completecases(lawsch85)
complete_cases2 = count(lsat_compl_cases2)
println("complete_cases2 = $complete_cases2")
```

Output of Script 9.6: `Missings.jl`

```
preview:
10×2 DataFrame
 Row | lsat     missLSAT
     | Int64?   Bool
     ---------------------------------------------
   1 |     156     false
   2 |     159     false
   3 |     157     false
   4 |     167     false
   5 | missing      true
   6 |     158     false
   7 |     155     false
   8 |     157     false
   9 | missing      true
  10 |     163     false

tot_missing = 6

tot_nonmissings = 150

freq_missLSAT_preview:
1×9 DataFrame
 Row | rank   salary  cost   LSAT   GPA    libvol  faculty  age    clsize
     | Int64  Int64   Int64  Int64  Int64  Int64   Int64    Int64  Int64
     ---------------------------------------------
   1 |     0       8      6      6      7       1        4     45       3

complete_cases1 = 90

complete_cases2 = 90
```

The question how to deal with missing values is not trivial and depends on many things. For basic functions such as the **mean** function, we cannot calculate the average, if at least one value is missing. Instead we have to use the function **skipmissing** as demonstrated in Script 9.5 (`NaN-Inf-Missing.jl`).

The regression command **lm** removes observations with missings by default considering each variable used in the specified regression model. The number of used observations can be extracted with the function **nobs**. This shows that it is usually a good idea to check the behavior of each function in the presence of missing data to avoid errors. Script 9.7 (`Missings-Analyses.jl`) gives examples of these features.

Script 9.7: `Missings-Analyses.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Statistics

lawsch85 = DataFrame(wooldridge("lawsch85"))

# missings:
x = lawsch85.LSAT
x_bar1 = mean(x)
x_bar2 = mean(skipmissing(x))
println("x_bar1 = $x_bar1\n")
println("x_bar2 = $x_bar2\n")

# observations and variables:
nrows = nrow(lawsch85)
ncols = ncol(lawsch85)
println("nrows = $nrows\n")
println("ncols = $ncols\n")

# regression (missings are taken care of by default):
reg = lm(@formula(log(salary) ~ LSAT + cost + age), lawsch85)
n = nobs(reg)
println("n = $n")
```

Output of Script 9.7: `Missings-Analyses.jl`

```
x_bar1 = missing

x_bar2 = 158.29333333333332

nrows = 156

ncols = 21

n = 95.0
```

9.4. Outlying Observations

Wooldridge (2019, Section 9.5) offers a very useful discussion of outlying observations. One of the important messages from the discussion is that dealing with outliers is a tricky business. To calculate studentized residuals discussed there, we follow the following steps: For each observation $i = 1, \cdots, n$, a regression model is estimated, where you use the usual y and x variables. You also use an additional explanatory dummy variable d_i, which is 1 for the i-th observation, and compute the studentized residual as the t statistic of d_i.

For the 5-th observation, for example, you estimate the model:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \cdots + \beta_k x_k + d_5 + u$$

The t statistic of d_5 is the studentized residual for observation $i = 5$, which describes the impact of that observation on the regression without using that observation. For the R&D example from Wooldridge (2019), Script 9.8 (`Outliers.jl`) calculates them in a loop and reports the highest and the lowest number. It also generates the histogram with overlayed density plot in Figure 9.1. Especially the highest value of 4.55 appears to be an extremely outlying value.

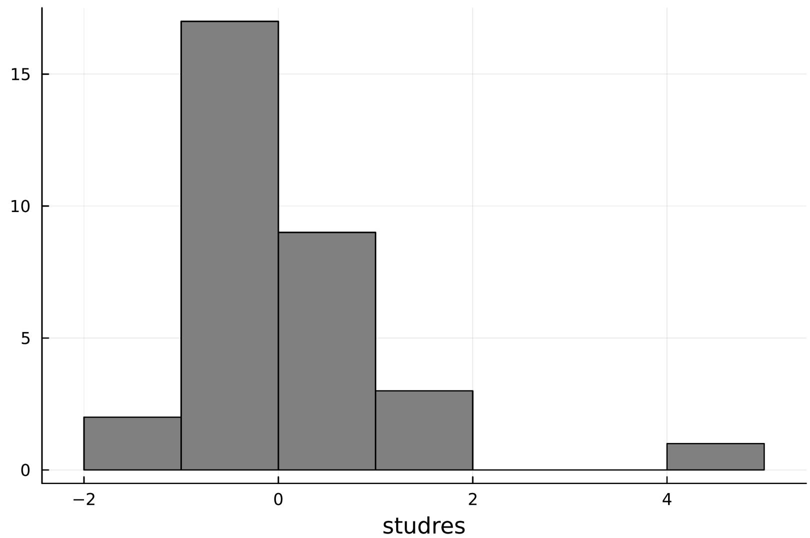
Script 9.8: `Outliers.jl`

```
using WooldridgeDatasets, GLM, DataFrames, LinearAlgebra, Plots

rdchem = DataFrame(wooldridge("rdchem"))

# create dummys for each observation with an identity matrix:
n = nrow(rdchem)
dummys = DataFrame(Matrix(1I, n, n), Symbol.(:d, 1:n)) # colnames d1, ..., d32

# studentized residuals for all observations:
studres = zeros(n)
for i in 1:n
    rdchem.di = dummys[:, i]
    reg_i = lm(@formula(rdintens ~ sales + profmarg + di), rdchem)
    # save t statistic (3rd column) of di (4th element):
    studres[i] = coeftable(reg_i).cols[3][4]
end

# display extreme values:
studres_max = maximum(studres)
studres_min = minimum(studres)
println("studres_max = $studres_max\n")
println("studres_min = $studres_min")

# histogram:
histogram(studres, color="grey", legend=false)
xlabel!("studres")
savefig("JlGraphs/Outliers.pdf")
```

Output of Script 9.8: `Outliers.jl`

```
studres_max = 4.555033421514251

studres_min = -1.8180393952811695
```

Figure 9.1. Outliers: Distribution of Studentized Residuals



9.5. Least Absolute Deviations (LAD) Estimation

As an alternative to OLS, the least absolute deviations (LAD) estimator is less sensitive to outliers. Instead of minimizing the sum of *squared* residuals, it minimizes the sum of the *absolute values* of the residuals.

Wooldridge (2019, Section 9.6) explains that the LAD estimator attempts to estimate the parameters of the conditional median $\text{Med}(y|x_1, \ldots, x_k)$ instead of the conditional mean $\text{E}(y|x_1, \ldots, x_k)$. This makes LAD a special case of quantile regression which studies general quantiles of which the median (=0.5 quantile) is just a special case. In the package **QuantileRegressions**, general quantile regression (and LAD as the special case) can easily be implemented with the command **qreg**.[2] It works very similar to **lm** for OLS estimation.

Script 9.9 (`LAD.jl`) demonstrates its application using the example from Wooldridge (2019, Example 9.8) and Script 9.8. Note that LAD inferences are only valid asymptotically, so the results in this example with $n = 32$ should be taken with a grain of salt.

Script 9.9: `LAD.jl`

```
using WooldridgeDatasets, DataFrames, GLM, QuantileRegressions

rdchem = DataFrame(wooldridge("rdchem"))

# OLS regression:
reg_ols = lm(@formula(rdintens ~ sales / 1000 + profmarg), rdchem)
table_reg_ols = coeftable(reg_ols)
println("table_reg_ols: \n$table_reg_ols\n")

# LAD regression:
reg_lad = qreg(@formula(rdintens ~ sales / 1000 + profmarg), rdchem, 0.5)
table_reg_lad = coeftable(reg_lad)
println("table_reg_lad: \n$table_reg_lad")
```

Output of Script 9.9: `LAD.jl`

```
table_reg_ols:

                  Coef.  Std. Error     t  Pr(>|t|)   Lower 95%  Upper 95%

(Intercept)   2.62526     0.585533   4.48    0.0001   1.42771     3.82281
sales / 1000  0.0533782   0.0440745  1.21    0.2356  -0.0367642   0.143521
profmarg      0.0446166   0.0461805  0.97    0.3420  -0.0498332   0.139066


table_reg_lad:

              Quantile  Estimate  Std.Error   t value

(Intercept)        0.5  1.62309   0.701203   2.31472
sales / 1000       0.5  0.018627  0.0527813  0.352909
profmarg           0.5  0.117905  0.0553034  2.13196
```

[2]For more information about the package, see Taylor (2006).

Part II.

Regression Analysis with Time Series Data

10. Basic Regression Analysis with Time Series Data

Time series differ from cross-sectional data in that each observation (i.e. row in a data frame) corresponds to one point or period in time. Section 10.1 introduces the most basic static time series models. In Section 10.2, we look into more technical details how to deal with time series data in *Julia*. Other aspects of time series models such as dynamics, trends, and seasonal effects are treated in Section 10.3.

10.1. Static Time Series Models

Static time series regression models describe the contemporaneous relation between the dependent variable y and the regressors $z_1, \ldots, z_k$. For each observation $t = 1, \ldots, n$, a static equation has the form

$$y_t = \beta_0 + \beta_1 z_{1t} + \cdots + \beta_k z_{kt} + u_t. \tag{10.1}$$

For the estimation of these models, the fact that we have time series does not make any practical difference. We can still use **lm** from **GLM** to estimate the parameters and the other tools for statistical inference. We only have to be aware that the assumptions needed for unbiased estimation and valid inference differ somewhat. Important differences to cross-sectional data are that we have to assume *strict* exogeneity (Assumption TS.3) for unbiasedness and no serial correlation (Assumption TS.5) for the usual variance-covariance formula to be valid, see Wooldridge (2019, Section 10.3).

Wooldridge, Example 10.2: Effects of Inflation and Deficits on Interest Rates

The data set `INTDEF` contains yearly information on interest rates and related time series between 1948 and 2003. Script 10.1 (`Example-10-2.jl`) estimates a static model explaining the interest rate **i3** with the inflation rate **inf** and the federal budget deficit **def**. There is nothing different in the implementation than for cross-sectional data. Both regressors are found to have a statistically significant relation to the interest rate.

Script 10.1: `Example-10-2.jl`

```
using WooldridgeDatasets, GLM, DataFrames

intdef = DataFrame(wooldridge("intdef"))

# linear regression of static model:
reg = lm(@formula(i3 ~ inf + def), intdef)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Output of Script 10.1: `Example-10-2.jl`

```
table_reg:

               Coef.  Std. Error     t  Pr(>|t|)  Lower 95%  Upper 95%

(Intercept)  1.73327    0.431967   4.01    0.0002   0.86685    2.59968
inf          0.605866   0.0821348  7.38    <1e-08   0.441124   0.770607
def          0.513058   0.118384   4.33    <1e-04   0.27561    0.750506
```

10.2. Time Series Data Types in *Julia*

For calculations specific to times series such as lags, trends, and seasonal effects, we will have to explicitly define the structure of our data. In *Julia*, there are several variable types specific to time series. The most important distinction is whether or not the data are equispaced. The observations of **equispaced** time series are collected at regular points in time. Typical examples are monthly, quarterly, or yearly data.

Observations of **irregular** time series have varying distances. An important example are daily financial data which are unavailable on weekends and bank holidays. Another example are financial tick data which contain a record each time a trade is completed which obviously does not happen at regular points in time. Although we will mostly work with equispaced data, we will briefly introduce these types in Section 10.2.2.

10.2.1. Equispaced Time Series in *Julia*

A convenient way to deal with equispaced time series in linear regression models, is to store them as a data frame (i.e. the type **`DataFrame`**). To capture the time dimension, you define an appropriate variable. With equispaced time series this is especially convenient with the data types **`Date`** and **`DateTime`** from the package **`Dates`**.[1] In combination with the function **`range`** they can be used to describe the time structure of equispaced data. It has the two positional arguments **`start`** and **`stop`**, and the two keyword arguments **`length`** and **`step`**:

- **`start`** / **`stop`**: Left/ right bound of first/ last observation is created by providing days, months, days, hours, minutes, seconds, and milliseconds in this order in the following format:

  ```
  DateTime(1978, 2, 1, 14, 35, 59, 2)
  ```

  If you are only interested in days, months and days, you can also use the **`date`** type with the same syntax. Other input formats to create date types can also be provided by the argument **`dateformat`**, so the following two lines are equivalent:

  ```
  Date(1978, 2, 1)
  Date("1978-02-01", dateformat="y-m-d")
  ```

- **`length`**: Number of equispaced points in time you need to generate.
- **`step`**: Number of observations per time unit. Examples:
  - **`step=Year(x)`**: Yearly data (repeating every **`x`** years)
  - **`step=Quarter(x)`**: Quarterly data (repeating every **`x`** quarters)
  - **`step=Month(x)`**: Monthly data (repeating every **`x`** months)

[1]For more information and useful examples, see `https://docs.julialang.org/en/v1/stdlib/Dates/`.

Because the data are equispaced, you have to specify three arguments and the remaining one is implied.

If you just need the **start**, **stop** and **step** argument, you can also use the syntax **start:step:stop** to create ranges. Therefore, the following two lines are equivalent:

```
range(Date(1978, 2, 1), Date(1978, 10, 1), step=Month(1))
Date(1978, 2, 1):Month(1):Date(1978, 10, 1)
```

It only makes sense to add these kind of variables to a data frame, if two consecutive rows represent two consecutive points in time in an ascending order.

As an example, consider the data set named BARIUM. It contains monthly data on imports of barium chloride from China between February 1978 and December 1988. Wooldridge (2019, Example 10.5) explains the data and background. Script 10.2 (Example-Barium.jl) demonstrates the use of **range** and how Figure 10.1 was generated. The time axis is automatically formatted appropriately.

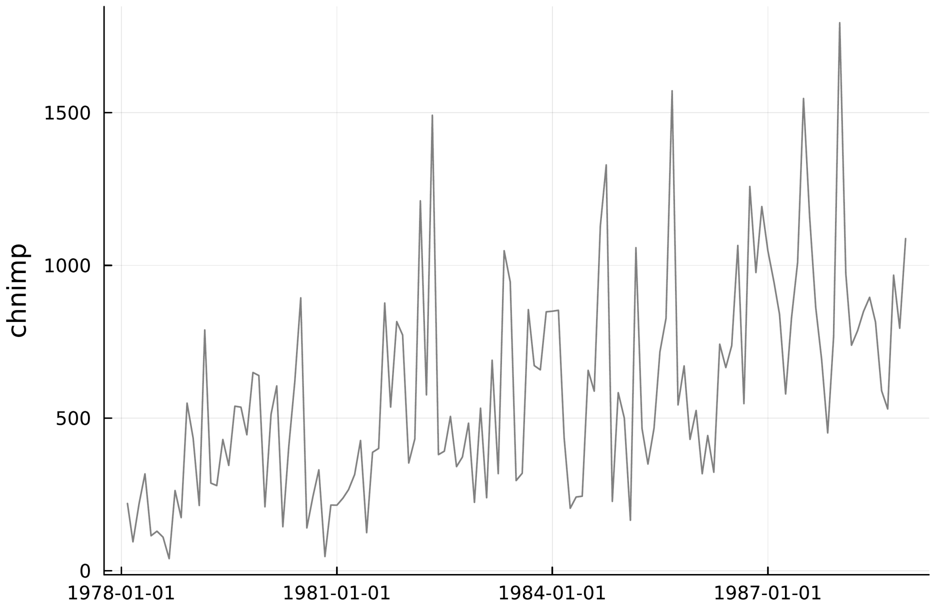
Script 10.2: Example-Barium.jl

```
using WooldridgeDatasets, DataFrames, Dates, Plots

barium = DataFrame(wooldridge("barium"))
T = nrow(barium)

# monthly time series starting Feb. 1978:
barium.date = range(Date(1978, 2, 1), step=Month(1), length=T)
preview = barium[1:5, ["date", "chnimp"]]
println("preview: \n$preview")

# plot chnimp:
plot(barium.date, barium.chnimp, legend=false, color="grey")
ylabel!("chnimp")
savefig("JlGraphs/Example-Barium.pdf")
```

Output of Script 10.2: Example-Barium.jl

```
preview:
5×2 DataFrame
 Row | date        chnimp
     | Date        Float64
     ---------------------------------------------
   1 | 1978-02-01  220.462
   2 | 1978-03-01   94.798
   3 | 1978-04-01  219.357
   4 | 1978-05-01  317.422
   5 | 1978-06-01  114.639
```

Figure 10.1. Time Series Plot: Imports of Barium Chloride from China



10.2.2. Irregular Time Series in *Julia*

For the remainder of this book, we will work with equispaced time series. But since irregular time series are important for example in finance, we will briefly introduce them here. The only thing changing is that you cannot use **range** to generate time stamps. Instead, these should be provided in your data.

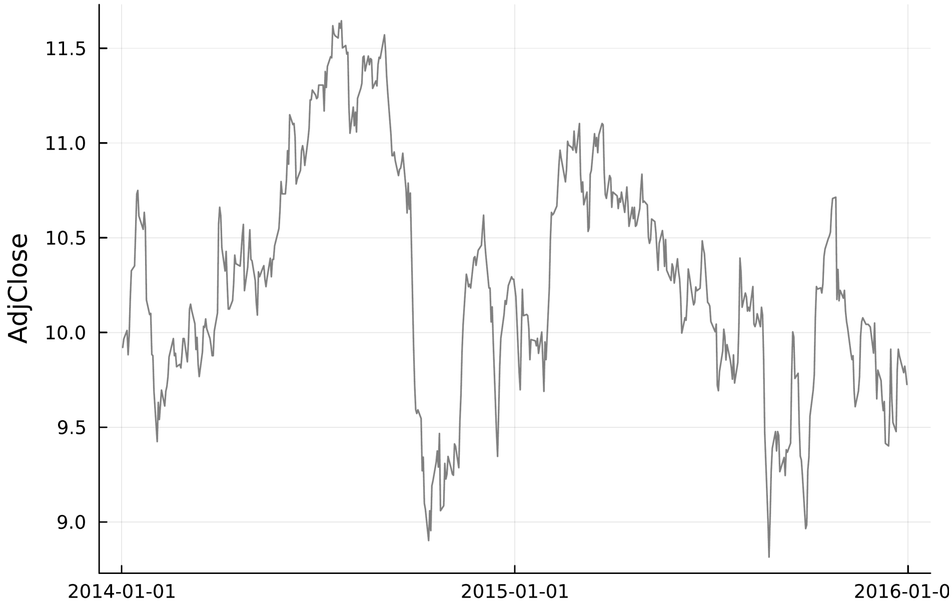
Daily financial data sets are important examples of irregular time series. Because of weekends and bank holidays, these data are not equispaced and each data point contains a time stamp - usually the date. To demonstrate this, we will briefly look at the package **MarketData** introduced in Section 1.3.3. It can automatically download financial data from Yahoo Finance and other sources. In order to do so, we must know the ticker symbol of the stock or whatever we are interested in. It can be looked up at `https://finance.yahoo.com/lookup`.

For example, the symbol for the Dow Jones Industrial Average is `^DJI`, Apple stocks have the symbol `AAPL` and the Ford Motor Company is simply abbreviated as `F`. Script 10.3 (`Example-StockData.jl`) demonstrates the import and the format of the imported data. They include information on opening, closing, high, and low prices as well as the trading volume and the adjusted (for events like stock splits and dividend payments) closing prices. We also print the first and last 5 rows of data, and plot the adjusted closing prices over time.

Script 10.3: `Example-StockData.jl`

```
using DataFrames, Dates, MarketData, Plots

# download data for "F" (= Ford Motor Company) and define start and end:
ticker = "F"
start_date = DateTime(2014, 1, 1)
end_date = DateTime(2016, 1, 1)

# import data:
F_data = yahoo(ticker, YahooOpt(period1=start_date, period2=end_date))

# look at imported data:
F_data_head = first(DataFrame(F_data), 5)
println("F_data_head: \n$F_data_head\n")

F_data_tail = last(DataFrame(F_data), 5)
println("F_data_tail: \n$F_data_tail")

# time series plot of adjusted closing prices:
plot(F_data.AdjClose, legend=false, color="grey")
ylabel!("AdjClose")
savefig("JlGraphs/Example-StockData.pdf")
```

Figure 10.2. Time Series Plot: Stock Prices of Ford Motor Company



Output of Script 10.3: `Example-StockData.jl`

```
F_data_head:
5×7 DataFrame
 Row | timestamp   Open     High     Low      Close    AdjClose  Volume
     | Date        Float64  Float64  Float64  Float64  Float64   Float64
     ---------------------------------------------
   1 | 2014-01-02    15.42    15.45    15.28    15.44   9.921    3.15285e7
   2 | 2014-01-03    15.52    15.64    15.3     15.51   9.96598  4.61223e7
   3 | 2014-01-06    15.72    15.76    15.52    15.58  10.011    4.26576e7
   4 | 2014-01-07    15.73    15.74    15.35    15.38   9.88244  5.44763e7
   5 | 2014-01-08    15.6     15.71    15.51    15.54   9.98525  4.84483e7

F_data_tail:
5×7 DataFrame
 Row | timestamp   Open     High     Low      Close    AdjClose  Volume
     | Date        Float64  Float64  Float64  Float64  Float64   Float64
     ---------------------------------------------
   1 | 2015-12-24    14.35    14.37    14.25    14.31   9.87743  9.0001e6
   2 | 2015-12-28    14.28    14.34    14.16    14.18   9.7877   1.36975e7
   3 | 2015-12-29    14.28    14.3     14.15    14.23   9.82221  1.88678e7
   4 | 2015-12-30    14.23    14.26    14.12    14.17   9.7808   1.38003e7
   5 | 2015-12-31    14.14    14.16    14.04    14.09   9.72558  1.9881e7
```

10.3. Other Time Series Models

10.3.1. Finite Distributed Lag Models

Finite distributed lag (FDL) models allow past values of regressors to affect the dependent variable. A FDL model of order q with an independent variable z can be written as

$$y_t = \alpha_0 + \delta_0 z_t + \delta_1 z_{t-1} + \cdots + \delta_q z_{t-q} + u_t. \tag{10.2}$$

Wooldridge (2019, Section 10.2) discusses the specification and interpretation of such models. For the implementation, we generate the q additional variables that reflect the lagged values $z_{t-1}, \ldots, z_{t-q}$ and include them in the model formula of **lm**. The function **lag(z, k)** allows to generate the lagged variable z_{t-k}.[2] Be aware that this only works if rows are sorted in an ascending order by the time variable. If your data frame **df** looks different and **time** is the time variable, you have to run **sort!(df,[:time])** first.

Wooldridge, Example 10.4: Effects of Personal Exemption on Fertility Rates

The data set FERTIL3 contains yearly information on the general fertility rate **gfr** and the personal tax exemption **pe** for the years 1913 through 1984. Dummy variables for the second world war **ww2** and the availability of the birth control pill **pill** are also included. Script 10.4 (Example-10-4.jl) shows the distributed lag model including contemporaneous **pe** and two lags. All **pe** coefficients are insignificantly different from zero according to the respective t tests. In Script 10.5 (Example-10-4-cont.jl) a usual F test implemented with **ftest** reveals that they are jointly significantly different from zero at a significance level of $\alpha = 5\%$ with a p value of 0.012 (see **fpval**). As Wooldridge (2019) discusses, this points to a multicollinearity problem.

Script 10.4: **Example-10-4.jl**

```
using WooldridgeDatasets, GLM, DataFrames

fertil3 = DataFrame(wooldridge("fertil3"))

# add all lags of pe up to order 2:
fertil3.pe_lag1 = lag(fertil3.pe, 1)
fertil3.pe_lag2 = lag(fertil3.pe, 2)

# linear regression of model with lags:
reg = lm(@formula(gfr ~ pe + pe_lag1 + pe_lag2 + ww2 + pill), fertil3)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

[2] We have encountered some problems with the **lag** function on different systems. You may load the **ShiftedArrays** package explicitly and then use the function **ShiftedArrays.lag**.

Output of Script 10.4: `Example-10-4.jl`

```
table_reg:

                  Coef.  Std. Error      t  Pr(>|t|)   Lower 95%   Upper 95%

(Intercept)   95.8705       3.28196   29.21    <1e-37   89.314     102.427
pe             0.0726718    0.125533   0.58    0.5647   -0.178109    0.323453
pe_lag1       -0.00577958   0.155663  -0.04    0.9705   -0.316752    0.305193
pe_lag2        0.0338268    0.126257   0.27    0.7896   -0.218401    0.286055
ww2          -22.1265      10.732     -2.06    0.0433  -43.5661     -0.68692
pill         -31.305        3.98156   -7.86    <1e-10  -39.2591    -23.3509
```

The long-run propensity (LRP) of FDL models measures the cumulative effect of a change in the independent variable z on the dependent variable y over time and is simply equal to the sum of the respective parameters

$$\text{LRP} = \delta_0 + \delta_1 + \cdots + \delta_q.$$

We can calculate it directly from the estimated regression model. For testing whether it is different from zero, we follow the procedure suggested by Wooldridge (2019): for $q = 2$, we substitute $\delta_0 = \text{LRP} - \delta_1 - \delta_2$ into the model allowing to test $H_0 : \text{LRP} = 0$ with a t test. For more details, see Wooldridge (2019, Section 10.4) and Script 10.5 (`Example-10-4-cont.jl`).

Wooldridge, Example 10.4: (continued)

Script 10.5 (`Example-10-4-cont.jl`) calculates the estimated LRP to be around 0.1.
To test its significance we modify the model

$$gfr_t = \alpha_0 + \delta_0 \cdot pe_t + \delta_1 \cdot pe_{t-1} + \delta_2 \cdot pe_{t-2} + \beta_1 \cdot ww2_t + \beta_2 \cdot pill_t$$

by substituting $\delta_0 = \text{LRP} - \delta_1 - \delta_2$. The resulting model is

$$gfr_t = \alpha_0 + \text{LRP} \cdot pe_t + \delta_1 \cdot \widetilde{pe}_{t-1} + \delta_2 \cdot \widetilde{pe}_{t-2} + \beta_1 \cdot ww2_t + \beta_2 \cdot pill_t$$

with the regressors $\widetilde{pe}_{t-1} = pe_{t-1} - pe_t$ and $\widetilde{pe}_{t-2} = pe_{t-2} - pe_t$. We can test $H_0 : \text{LRP} = 0$ by looking at the respective t test. As a result LRP is significantly different from zero with a p value of around 0.0012.

Script 10.5: Example-10-4-cont.jl

```
using WooldridgeDatasets, GLM, DataFrames, Distributions

fertil3 = DataFrame(wooldridge("fertil3"))

# add all lags of pe up to order 2:
fertil3.pe_lag1 = lag(fertil3.pe, 1)
fertil3.pe_lag2 = lag(fertil3.pe, 2)

# handle missings due to lagged data manually (important for ftest):
fertil3 = fertil3[Not([1, 2]), :]

# linear regression of model with lags:
reg_ur = lm(@formula(gfr ~ pe + pe_lag1 + pe_lag2 + ww2 + pill), fertil3)

# F test (H0: all pe coefficients are zero):
reg_r = lm(@formula(gfr ~ ww2 + pill), fertil3)
ftest_res = ftest(reg_r.model, reg_ur.model)
fstat = ftest_res.fstat[2]
fpval = ftest_res.pval[2]
println("fstat = $fstat\n")
println("fpval = $fpval\n")

# calculating the LRP:
b_pe_tot = sum(coef(reg_ur)[[2, 3, 4]])
println("b_pe_tot = $b_pe_tot\n")

# testing LRP=0:
fertil3.ptm1pt = fertil3.pe_lag1 - fertil3.pe
fertil3.ptm2pt = fertil3.pe_lag2 - fertil3.pe
reg_LRP = lm(@formula(gfr ~ pe + ptm1pt + ptm2pt + ww2 + pill), fertil3)
table_res_LRP = coeftable(reg_LRP)
println("table_res_LRP: \n$table_res_LRP")
```

Output of Script 10.5: Example-10-4-cont.jl

```
fstat = 3.9729640469785976

fpval = 0.011652005303125688

b_pe_tot = 0.10071909027975678

table_res_LRP:

                    Coef.  Std. Error      t  Pr(>|t|)    Lower 95%   Upper 95%

(Intercept)   95.8705       3.28196    29.21    <1e-37   89.314      102.427
pe             0.100719     0.0298027   3.38    0.0012    0.0411814    0.160257
ptm1pt        -0.00577958   0.155663   -0.04    0.9705   -0.316752     0.305193
ptm2pt         0.0338268    0.126257    0.27    0.7896   -0.218401     0.286055
ww2          -22.1265      10.732      -2.06    0.0433  -43.5661      -0.68692
pill         -31.305        3.98156    -7.86    <1e-10  -39.2591     -23.3509
```

10.3.2. Trends

As pointed out by Wooldridge (2019, Section 10.5), deterministic linear (and exponential) time trends are accounted for by adding the time measure as another independent variable.

Wooldridge, Example 10.7: Housing Investment and Prices

The data set HSEINV provides annual observations on housing investments **invpc** and housing prices **price** for the years 1947 through 1988. Using a double-logarithmic specification, Script 10.6 (Example-10-7.jl) estimates a regression model with and without a linear trend. The variable **t** is used to capture the time trend in the second regression. Forgetting to add the trend leads to the spurious finding that investments and prices are related.
Because of the logarithmic dependent variable, the trend in **invpc** (as opposed to log **invpc**) is exponential. The estimated coefficient implies a 1% yearly increase in investments.

Script 10.6: Example-10-7.jl

```
using WooldridgeDatasets, GLM, DataFrames

hseinv = DataFrame(wooldridge("hseinv"))

# linear regression without time trend:
reg_wot = lm(@formula(log(invpc) ~ log(price)), hseinv)
table_reg_wot = coeftable(reg_wot)
println("table_reg_wot: \n$table_reg_wot\n")

# linear regression with time trend (data set includes a time variable t):
reg_wt = lm(@formula(log(invpc) ~ log(price) + t), hseinv)
table_reg_wt = coeftable(reg_wt)
println("table_reg_wt: \n$table_reg_wt")
```

Output of Script 10.6: Example-10-7.jl

```
table_reg_wot:

                 Coef.  Std. Error       t  Pr(>|t|)  Lower 95%  Upper 95%

(Intercept)  -0.550235   0.0430266  -12.79    <1e-14  -0.637195  -0.463275
log(price)    1.24094    0.382419     3.24    0.0024   0.468045   2.01384


table_reg_wt:

                   Coef.  Std. Error      t  Pr(>|t|)    Lower 95%   Upper 95%

(Intercept)  -0.91306     0.135613    -6.73    <1e-07  -1.18736     -0.638756
log(price)   -0.380961    0.678835    -0.56    0.5779  -1.75404      0.992113
t             0.00982873  0.00351221   2.80    0.0079   0.00272461   0.0169328
```

10.3.3. Seasonality

To account for seasonal effects, we add dummy variables for all but one (the reference) "season". So with monthly data, we can include eleven dummies, see Chapter 7 for a detailed discussion.

Wooldridge, Example 10.11: Effects of Antidumping Filings

The data in BARIUM were used in an antidumping case. They are monthly data on barium chloride imports from China between February 1978 and December 1988. Wooldridge (2019, Example 10.5) explains the data and background. When we estimate a model with monthly dummies, they do not have significant coefficients except the dummy for April which is marginally significant. An F test which is not reported reveals no joint significance.

Script 10.7: Example-10-11.jl

```
using WooldridgeDatasets, GLM, DataFrames

barium = DataFrame(wooldridge("barium"))

# linear regression with seasonal effects:
reg = lm(@formula(log(chnimp) ~ log(chempi) + log(gas) +
                                log(rtwex) + befile6 + affile6 + afdec6 +
                                feb + mar + apr + may + jun + jul +
                                aug + sep + oct + nov + dec), barium)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Output of Script 10.7: Example-10-11.jl

```
table_reg:

                  Coef.  Std. Error      t  Pr(>|t|)   Lower 95%   Upper 95%

(Intercept)  16.7792       32.4286     0.52    0.6059  -47.4678    81.0262
log(chempi)   3.26506       0.49293    6.62    <1e-08    2.28848    4.24165
log(gas)     -1.27814       1.38901   -0.92    0.3594   -4.03002    1.47374
log(rtwex)    0.663045      0.471304   1.41    0.1622   -0.270692   1.59678
befile6       0.139703      0.266808   0.52    0.6016   -0.388891   0.668297
affile6       0.0126324     0.278687   0.05    0.9639   -0.539496   0.564761
afdec6       -0.5213        0.30195   -1.73    0.0870   -1.11952    0.0769169
feb          -0.417711      0.304444  -1.37    0.1728   -1.02087    0.185448
mar           0.059052      0.264731   0.22    0.8239   -0.465427   0.583531
apr          -0.451483      0.268386  -1.68    0.0953   -0.983205   0.0802389
may           0.033309      0.269242   0.12    0.9018   -0.500109   0.566727
jun          -0.206332      0.269252  -0.77    0.4451   -0.739767   0.327104
jul           0.00383659    0.278767   0.01    0.9890   -0.54845    0.556124
aug          -0.157064      0.277993  -0.56    0.5732   -0.707818   0.393689
sep          -0.134161      0.267656  -0.50    0.6172   -0.664435   0.396114
oct           0.0516925     0.266851   0.19    0.8467   -0.476988   0.580373
nov          -0.24626       0.262827  -0.94    0.3508   -0.766968   0.274448
dec           0.132838      0.271423   0.49    0.6255   -0.404901   0.670576
```

11. Further Issues in Using OLS with Time Series Data

This chapter introduces important concepts for time series analyses. Section 11.1 discusses the general conditions under which asymptotic analyses work with time series data. An important requirement will be that the time series exhibit weak dependence. In Section 11.2, we study highly persistent time series and present some simulation exercises. One solution to this problem is first differencing as demonstrated in Section 11.3. How this can be done in the regression framework is the topic of Section 11.4.

11.1. Asymptotics with Time Series

As Wooldridge (2019, Section 11.2) discusses, asymptotic arguments also work with time series data under certain conditions. Importantly, we have to assume that the data are stationary and weakly dependent (Assumption TS.1). On the other hand, we can relax the strict exogeneity assumption TS.3 and only have to assume contemporaneous exogeneity (Assumption TS.3'). Under the appropriate set of assumptions, we can use standard OLS estimation and inference.

Wooldridge, Example 11.4: Efficient Markets Hypothesis

The efficient markets hypothesis claims that we cannot predict stock returns from past returns. In a simple AR(1) model in which returns are regressed on lagged returns, this would imply a population slope coefficient of zero. The data set `NYSE` contains data on weekly stock returns and we use the function **`lag(data, k)`** to compute the k'th lag.
Script 11.1 (`Example-11-4.jl`) shows the analyses. Regression 1 is the AR(1) model also discussed by Wooldridge (2019). Models 2 and 3 add second and third lags to estimate higher-order AR(p) models. In all models, no lagged value has a significant coefficient and also the F tests for joint significance (not included in the script) do not reject the efficient markets hypothesis.

Script 11.1: `Example-11-4.jl`

```
using WooldridgeDatasets, GLM, DataFrames, RegressionTables

nyse = DataFrame(wooldridge("nyse"))
nyse.ret = nyse.return

# add all lags up to order 3:
nyse.ret_lag1 = lag(nyse.ret, 1)
nyse.ret_lag2 = lag(nyse.ret, 2)
nyse.ret_lag3 = lag(nyse.ret, 3)

# linear regression of model with lags:
reg1 = lm(@formula(ret ~ ret_lag1), nyse)
reg2 = lm(@formula(ret ~ ret_lag1 + ret_lag2), nyse)
reg3 = lm(@formula(ret ~ ret_lag1 + ret_lag2 + ret_lag3), nyse)

# print results with RegressionTables:
regtable(reg1, reg2, reg3)
```

Output of Script 11.1: `Example-11-4.jl`

```
-----------------------------------------
                           ret
              ---------------------------
                   (1)        (2)        (3)
-----------------------------------------
(Intercept)     0.180*     0.186*     0.179*
               (0.081)    (0.081)    (0.082)
ret_lag1         0.059      0.060      0.061
               (0.038)    (0.038)    (0.038)
ret_lag2                   -0.038     -0.040
                          (0.038)    (0.038)
ret_lag3                               0.031
                                     (0.038)
-----------------------------------------
Estimator          OLS        OLS        OLS
-----------------------------------------
N                  689        688        687
R2               0.003      0.005      0.006
-----------------------------------------
```

We can do a similar analysis for daily data. The package **MarketData** introduced in Section 1.3.3 allows us to directly download daily stock prices from Yahoo Finance. Script 11.2 (`Example-EffMkts.jl`) downloads daily stock prices of Apple (ticker symbol `AAPL`) and stores them as a **DataFrame** object. From the prices p_t, daily returns r_t are calculated using the standard formula

$$r_t = \log(p_t) - \log(p_{t-1}) \approx \frac{p_t - p_{t-1}}{p_{t-1}}.$$

Note that in the script, we calculate the difference using the function **diff**. It calculates the difference from trading day to trading day, ignoring the fact that some of them are separated by weekends or holidays. Obviously, this procedure only works, if two consecutive rows represent two consecutive points in time. Figure 11.1 plots the returns of the Apple stock. Even though we now have $n = 2267$ observations of daily returns, we cannot find any relation between current and past returns which supports (this version of) the efficient markets hypothesis.

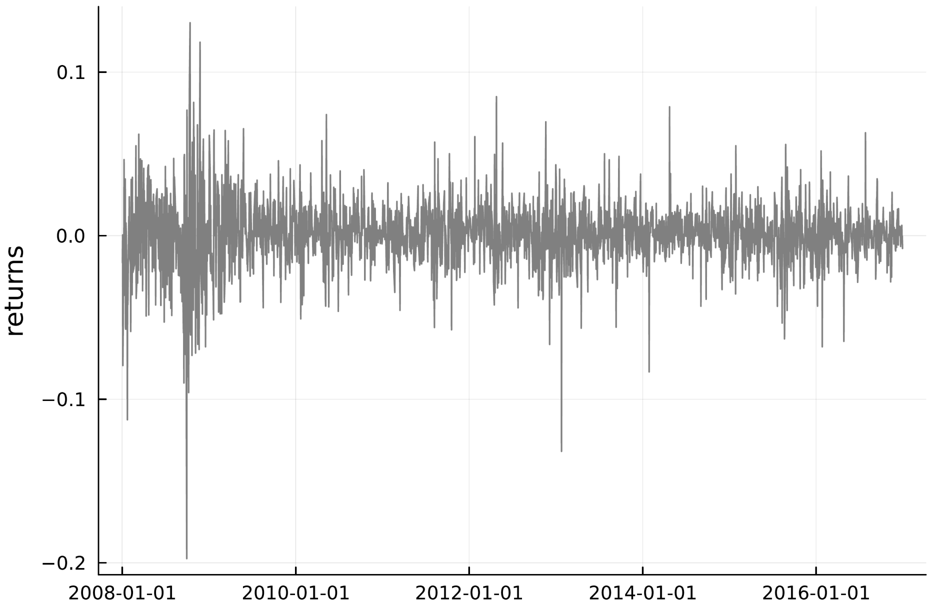
Script 11.2: Example-EffMkts.jl

```
using DataFrames, GLM, Dates, MarketData, Plots, RegressionTables

# download data for "AAPL" (= Apple) and define start and end:
ticker = "AAPL"
start_date = DateTime(2007, 12, 31)
end_date = DateTime(2017, 01, 01)

# import data as DataFrame:
AAPL_data = DataFrame(yahoo(ticker,
    YahooOpt(period1=start_date, period2=end_date)))

# calculate return as the difference of logged prices:
AAPL_data.ret = vcat(missing, diff(log.(AAPL_data.AdjClose)))

# time series plot of returns:
plot(AAPL_data.timestamp, AAPL_data.ret, legend=false, color="grey")
ylabel!("returns")
savefig("JlGraphs/Example-EffMkts.pdf")

# linear regression of models with lags:
AAPL_data.ret_lag1 = lag(AAPL_data.ret, 1)
AAPL_data.ret_lag2 = lag(AAPL_data.ret, 2)
AAPL_data.ret_lag3 = lag(AAPL_data.ret, 3)

reg1 = lm(@formula(ret ~ ret_lag1), AAPL_data)
reg2 = lm(@formula(ret ~ ret_lag1 + ret_lag2), AAPL_data)
reg3 = lm(@formula(ret ~ ret_lag1 + ret_lag2 + ret_lag3), AAPL_data)

# print results with RegressionTables:
regtable(reg1, reg2, reg3)
```

Output of Script 11.2: Example-EffMkts.jl

```
-----------------------------------------
                              ret
              ---------------------------
                    (1)        (2)        (3)
-----------------------------------------
(Intercept)      0.001      0.001      0.001
               (0.000)    (0.000)    (0.000)
ret_lag1        -0.003     -0.004     -0.003
               (0.021)    (0.021)    (0.021)
ret_lag2                   -0.029     -0.030
                          (0.021)    (0.021)
ret_lag3                               0.005
                                     (0.021)
-----------------------------------------
Estimator          OLS        OLS        OLS
-----------------------------------------
N                2,266      2,265      2,264
R2               0.000      0.001      0.001
-----------------------------------------
```

Figure 11.1. Time Series Plot: Daily Stock Returns 2008–2016, Apple Inc.



11.2. The Nature of Highly Persistent Time Series

The simplest model for highly persistent time series is a random walk. It can be written as

$$y_t = y_{t-1} + e_t \tag{11.1}$$
$$= y_0 + e_1 + e_2 + \cdots + e_{t-1} + e_t \tag{11.2}$$

where the shocks $e_1, \ldots, e_t$ are i.i.d. with a zero mean. It is a special case of a unit root process. Random walk processes are strongly dependent and nonstationary, violating assumption TS1′ required for the consistency of OLS parameter estimates. As Wooldridge (2019, Section 11.3) shows, the variance of y_t (conditional on y_0) increases linearly with t:

$$\text{Var}(y_t|y_0) = \sigma_e^2 \cdot t. \tag{11.3}$$

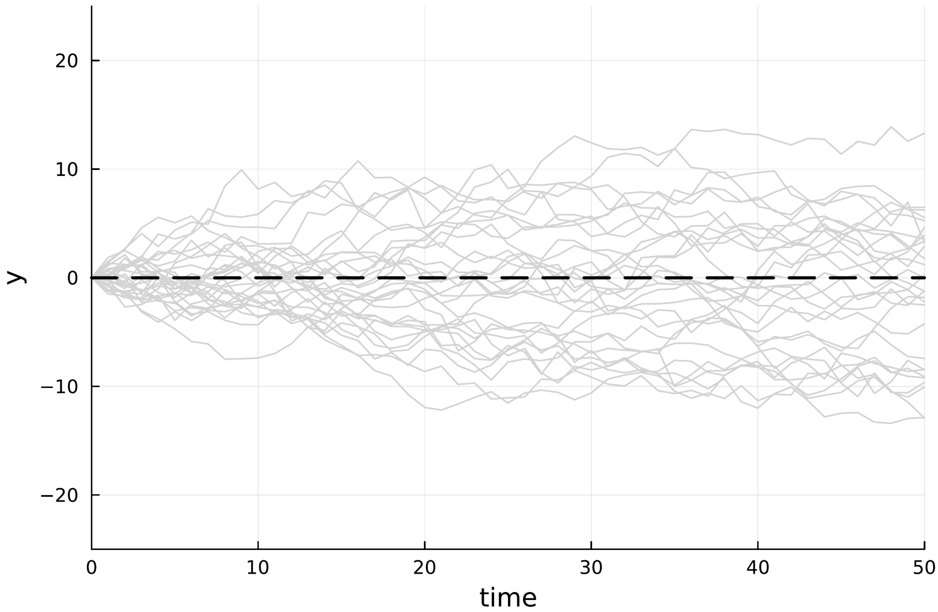
This can be easily seen in a simulation exercise. Script 11.3 (`Simulate-RandomWalk.jl`) draws 30 realizations from a random walk process with i.i.d. standard normal shocks e_t. After initializing the random number generator, an empty figure with the right dimensions is produced. Then, the realizations of the time series are drawn in a loop.[1] In each of the 30 draws, we first obtain a sample of the $n = 50$ shocks $e_1, \ldots, e_{50}$. The random walk is generated as the cumulative sum of the shocks according to Equation 11.2 with an initial value of $y_0 = 0$. The respective time series are then added to the plot. In the resulting Figure 11.2, the increasing variance can be seen easily.

Script 11.3: `Simulate-RandomWalk.jl`

```
using Random, Distributions, Statistics, Plots

# set the random seed:
Random.seed!(12345)

# initialize plot:
x_range = range(0, 50, 51)
plot(xlims=(0, 50), ylims=(-25, 25))

# loop over draws:
for r in 1:30
    # i.i.d. standard normal shock:
    e = rand(Normal(0, 1), 51)

    # set first entry to 0 (gives y_0 = 0):
    e[1] = 0

    # random walk as cumulative sum of shocks:
    y = cumsum(e)

    # add line to graph:
    plot!(x_range, y, color="lightgrey", legend=false)
end

hline!([0], color="black", linewidth=2, linestyle=:dash)
xlabel!("time")
ylabel!("y")
savefig("JlGraphs/Simulate-RandomWalk.pdf")
```

[1] For a review of random number generation, see Section 1.6.4.

Figure 11.2. Simulations of a Random Walk Process

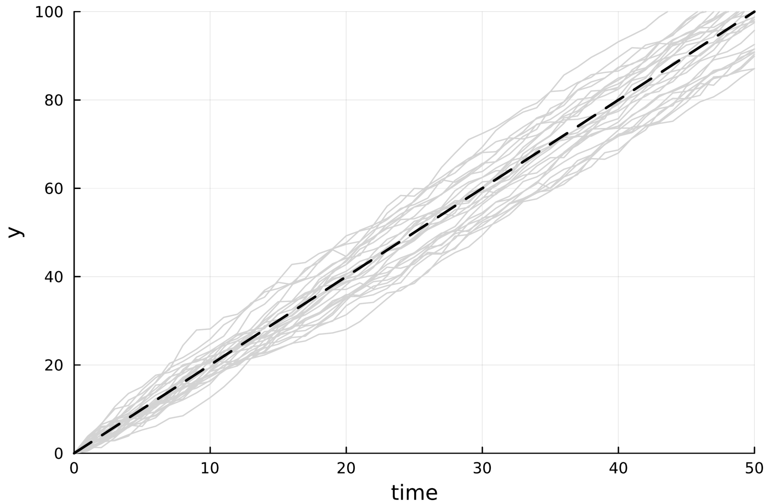


A simple generalization is a random walk with drift:

$$y_t = \alpha_0 + y_{t-1} + e_t \tag{11.4}$$
$$= y_0 + \alpha_0 \cdot t + e_1 + e_2 + \cdots + e_{t-1} + e_t. \tag{11.5}$$

Script 11.4 (`Simulate-RandomWalkDrift.jl`) simulates such a process with $\alpha_0 = 2$ and i.i.d. standard normal shocks e_t. The resulting time series are plotted in Figure 11.3. The values fluctuate around the expected value $\alpha_0 \cdot t$. But unlike weakly dependent processes, they do not tend towards their mean, so the variance increases like for a simple random walk process.

Figure 11.3. Simulations of a Random Walk Process with Drift



Script 11.4: `Simulate-RandomWalkDrift.jl`

```
using Random, Distributions, Statistics, Plots

# set the random seed:
Random.seed!(12345)

# initialize plot:
x_range = range(0, 50, 51)
plot(xlims=(0, 50), ylims=(0, 100))

# loop over draws:
for r in 1:30
    # i.i.d. standard normal shock:
    e = rand(Normal(0, 1), 51)

    # set first entry to 0 (gives y_0 = 0):
    e[1] = 0

    # random walk as cumulative sum of shocks:
    y = cumsum(e) + 2 * x_range

    # add line to graph:
    plot!(x_range, y, color="lightgrey", legend=false)
end

plot!(x_range, 2 * x_range, color="black", linewidth=2, linestyle=:dash)
xlabel!("time")
ylabel!("y")
savefig("JlGraphs/Simulate-RandomWalkDrift.pdf")
```

An obvious question is whether a given sample is from a unit root process such as a random walk. We will cover tests for unit roots in Section 18.2.

11.3. Differences of Highly Persistent Time Series

The simplest way to deal with highly persistent time series is to work with their differences rather than their levels. The first difference of the random walk with drift is:

$$y_t = \alpha_0 + y_{t-1} + e_t \tag{11.6}$$

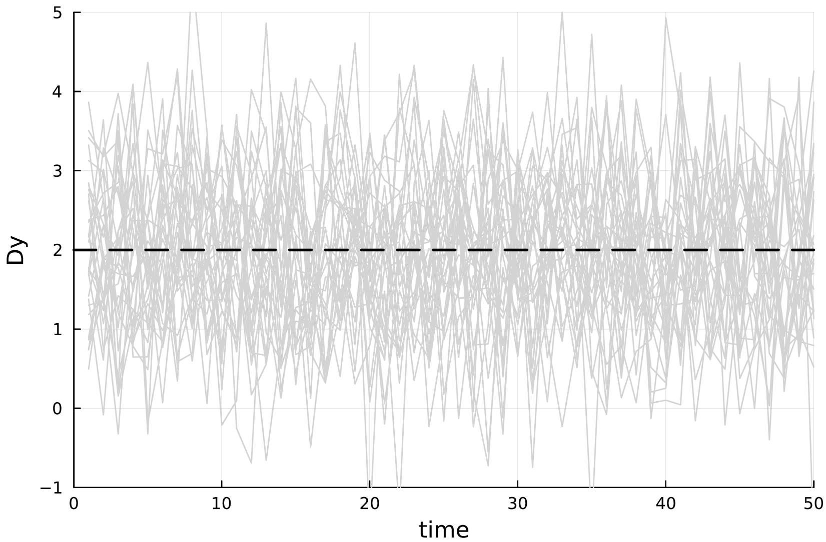
$$\Delta y_t = y_t - y_{t-1} = \alpha_0 + e_t \tag{11.7}$$

This is an i.i.d. process with mean α_0. Script 11.5 (`Simulate-RandomWalkDrift-Diff.jl`) repeats the same simulation as Script 11.4 (`Simulate-RandomWalkDrift.jl`) but calculates the differences using **`y[2:51] .- y[1:50]`**. From now on, we will use the more convenient function **`diff`** for the same task. The resulting series are shown in Figure 11.4. They have a constant mean of 2, a constant variance of $\sigma_e^2 = 1$, and are independent over time.

Script 11.5: `Simulate-RandomWalkDrift-Diff.jl`

```
using Random, Distributions, Statistics, Plots

# set the random seed:
Random.seed!(12345)

# initialize plot:
x_range = range(1, 50, 50)
plot(xlims=(0, 50), ylims=(-1, 5))

# loop over draws:
for r in 1:30
    # i.i.d. standard normal shock:
    e = rand(Normal(0, 1), 51)

    # set first entry to 0 (gives y_0 = 0):
    e[1] = 0

    # random walk as cumulative sum of shocks:
    y = cumsum(2 .+ e)

    # first difference:
    Dy = y[2:51] .- y[1:50]

    # add line to graph:
    plot!(x_range, Dy, color="lightgrey", legend=false)
end

hline!([2], color="black", linewidth=2, linestyle=:dash)
xlabel!("time")
ylabel!("Dy")
savefig("JlGraphs/Simulate-RandomWalkDrift-Diff.pdf")
```

Figure 11.4. Simulations of a Random Walk Process with Drift: First Differences



11.4. Regression with First Differences

Adding first differences to regression models is straightforward. You have to add the dependent or independent variable **var** as a first difference to your data before starting the usual **lm** command. The same holds, if you want to combine differences with lags in your specifications. This is demonstrated in Example 11.6.

As already mentioned, the functions **lag** and **diff** are helpful, but they require that consecutive rows represent two consecutive points in time. These commands do not use any time stamp you may have provided before.

Wooldridge, Example 11.6: Fertility Equation

We continue Example 10.4 and specify the fertility equation in first differences. Script 11.6 (`Example-11-6.jl`) shows the analyses. While the first difference of the tax exemptions has no significant effect, its second lag has a significantly positive coefficient in the second model. This is consistent with fertility reacting two years after a change of the tax code.

Script 11.6: `Example-11-6.jl`

```
using WooldridgeDatasets, GLM, DataFrames

fertil3 = DataFrame(wooldridge("fertil3"))

# compute first differences (first difference is always missing):
fertil3.gfr_diff1 = vcat(missing, diff(fertil3.gfr))
fertil3.pe_diff1 = vcat(missing, diff(fertil3.pe))
preview = fertil3[1:5, ["gfr", "gfr_diff1", "pe", "pe_diff1"]]
println("preview: \n$preview\n")

# linear regression of model with first differences:
reg1 = lm(@formula(gfr_diff1 ~ pe_diff1), fertil3)
table_reg1 = coeftable(reg1)
println("table_reg1: \n$table_reg1\n")

# linear regression of model with lagged differences:
fertil3.pe_diff1_lag1 = lag(fertil3.pe_diff1, 1)
fertil3.pe_diff1_lag2 = lag(fertil3.pe_diff1, 2)

reg2 = lm(@formula(gfr_diff1 ~ pe_diff1 + pe_diff1_lag1 + pe_diff1_lag2),
    fertil3)
table_reg2 = coeftable(reg2)
println("table_reg2: \n$table_reg2")
```

Output of Script 11.6: Example-11-6.jl

```
preview:
5×4 DataFrame
 Row | gfr      gfr_diff1  pe       pe_diff1
     | Float64  Float64?   Float64  Float64?
     ---------------------------------------------
   1 |   124.7  missing       0.0   missing
   2 |   126.6        1.9     0.0        0.0
   3 |   125.0       -1.6     0.0        0.0
   4 |   123.4       -1.6     0.0        0.0
   5 |   121.0       -2.4    19.27      19.27

table_reg1:

                 Coef.  Std. Error      t  Pr(>|t|)   Lower 95%  Upper 95%

(Intercept)  -0.78478     0.50204    -1.56    0.1226  -1.78632    0.216763
pe_diff1     -0.0426776   0.0283672  -1.50    0.1370  -0.0992686  0.0139134


table_reg2:

                   Coef.  Std. Error      t  Pr(>|t|)   Lower 95%   Upper 95%

(Intercept)    -0.963679    0.46776    -2.06    0.0434  -1.89786    -0.0294976
pe_diff1       -0.0362021   0.0267737  -1.35    0.1810  -0.089673    0.0172687
pe_diff1_lag1  -0.0139706   0.0275539  -0.51    0.6139  -0.0689997   0.0410584
pe_diff1_lag2   0.10999     0.0268797   4.09    0.0001   0.0563071   0.163672
```

12. Serial Correlation and Heteroscedasticity in Time Series Regressions

In Chapter 8, we discussed the consequences of heteroscedasticity in cross-sectional regressions. In the time series setting, similar consequences and strategies apply to both heteroscedasticity (with some specific features) and serial correlation of the error term. Unbiasedness and consistency of the OLS estimators are unaffected. But the OLS estimators are inefficient and the usual standard errors and inferences are invalid.

We first discuss how to test for serial correlation in Section 12.1. Section 12.2 introduces efficient estimation using feasible GLS estimators. As an alternative, we can still use OLS and calculate standard errors that are valid under both heteroscedasticity and autocorrelation as discussed in Section 12.3. Finally, Section 12.4 covers heteroscedasticity and autoregressive conditional heteroscedasticity (ARCH) models.

12.1. Testing for Serial Correlation of the Error Term

Suppose we are worried that the error terms $u_1, u_2, \ldots$ in a regression model of the form

$$y_t = \beta_0 + \beta_1 x_{t1} + \beta_2 x_{t2} + \cdots + \beta_k x_{tk} + u_t \tag{12.1}$$

are serially correlated. A straightforward and intuitive testing approach is described by Wooldridge (2019, Section 12.3). It is based on the fitted residuals $\hat{u}_t = y_t - \hat{\beta}_0 - \hat{\beta}_1 x_{t1} - \cdots - \hat{\beta}_k x_{tk}$ which can be obtained in **GLM** with the **residuals** function, see Section 2.2.

To test for AR(1) serial correlation under strict exogeneity, we regress $\hat{u}_t$ on their lagged values $\hat{u}_{t-1}$. If the regressors are not necessarily strictly exogenous, we can adjust the test by adding the original regressors $x_{t1}, \ldots, x_{tk}$ to this regression. Then we perform the usual t test on the coefficient of $\hat{u}_{t-1}$.

For testing for higher order serial correlation, we add higher order lags $\hat{u}_{t-2}, \hat{u}_{t-3}, \ldots$ as explanatory variables and test the joint hypothesis that they are all equal to zero using either an F test or a Lagrange multiplier (LM) test. Especially the latter version is often called Breusch-Godfrey test.

Wooldridge, Example 12.2: Testing for AR(1) Serial Correlation

We use this example to demonstrate the "pedestrian" way to test for autocorrelation which is actually straightforward and instructive. We estimate two versions of the Phillips curve: a static model

$$\texttt{inf}_t = \beta_0 + \beta_1 \texttt{unem}_t + u_t$$

and an expectation-augmented Phillips curve

$$\Delta\texttt{inf}_t = \beta_0 + \beta_1 \texttt{unem}_t + u_t.$$

Scripts 12.1 (`Example-12-2-Static.jl`) and 12.2 (`Example-12-2-ExpAug.jl`) show the analyses. After the estimation, the residuals are extracted by calling **`residuals`** and regressed on their lagged values. We report standard errors and t statistics. While there is strong evidence for autocorrelation in the static equation with a t statistic of $\frac{0.573}{0.116} \approx 4.93$, the null hypothesis of no autocorrelation cannot be rejected in the second model with a t statistic of $\frac{-0.036}{0.124} \approx -0.29$.

Script 12.1: `Example-12-2-Static.jl`

```
using WooldridgeDatasets, GLM, DataFrames

phillips = DataFrame(wooldridge("phillips"))
yt96 = subset(phillips, :year => ByRow(<=(1996)))

# estimation of static Phillips curve:
reg_s = lm(@formula(inf ~ unem), yt96)

# residuals and AR(1) test:
yt96.resid_s = residuals(reg_s)
yt96.resid_s_lag1 = lag(yt96.resid_s, 1)

reg = lm(@formula(resid_s ~ resid_s_lag1), yt96)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Output of Script 12.1: `Example-12-2-Static.jl`

```
table_reg:

                  Coef.  Std. Error      t  Pr(>|t|)  Lower 95%  Upper 95%

(Intercept)   -0.113397    0.359404  -0.32    0.7538  -0.836839   0.610046
resid_s_lag1   0.572969    0.116133   4.93    <1e-04   0.339205   0.806734
```

Script 12.2: `Example-12-2-ExpAug.jl`

```
using WooldridgeDatasets, GLM, DataFrames

phillips = DataFrame(wooldridge("phillips"))
yt96 = subset(phillips, :year => ByRow(<=(1996)))

# estimation of expectations-augmented Phillips curve:
yt96.inf_diff1 = vcat(missing, diff(yt96.inf))
yt96 = yt96[Not(1), :]
reg_ea = lm(@formula(inf_diff1 ~ unem), yt96)

yt96.resid_ea = residuals(reg_ea)
yt96.resid_ea_lag1 = lag(yt96.resid_ea, 1)

reg = lm(@formula(resid_ea ~ resid_ea_lag1), yt96)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Output of Script 12.2: `Example-12-2-ExpAug.jl`

```
table_reg:

                     Coef.  Std. Error      t  Pr(>|t|)  Lower 95%  Upper 95%

(Intercept)      0.194166     0.300384   0.65    0.5213  -0.410839   0.79917
resid_ea_lag1  -0.0355928     0.123891  -0.29    0.7752  -0.285122   0.213936
```

The F test of AR(q) serial correlation can be performed by the function **`ftest`** in **`GLM`** as demonstrated in the following example.

Wooldridge, Example 12.4: Testing for AR(3) Serial Correlation

We already used the monthly data set `BARIUM` and estimated a model for barium chloride imports in Example 10.11. Script 12.3 (`Example-12-4.jl`) estimates the model and tests for AR(3) serial correlation. This gives exactly the results reported by Wooldridge (2019).

Script 12.3: `Example-12-4.jl`

```
using WooldridgeDatasets, GLM, DataFrames

barium = DataFrame(wooldridge("barium"))

reg = lm(@formula(log(chnimp) ~ log(chempi) + log(gas) + log(rtwex) +
                                befile6 + affile6 + afdec6), barium)

# testing resid_lag1 = 0, resid_lag2 = 0 and resid_lag3 = 0:
barium.resid = residuals(reg)
barium.resid_lag1 = lag(barium.resid, 1)
barium.resid_lag2 = lag(barium.resid, 2)
barium.resid_lag3 = lag(barium.resid, 3)
barium = barium[Not(1:3), :]

reg_manual_ur = lm(@formula(resid ~ resid_lag1 + resid_lag2 + resid_lag3 +
                                    log(chempi) + log(gas) + log(rtwex) +
                                    befile6 + affile6 + afdec6), barium)
```

```
reg_manual_r = lm(@formula(resid ~ log(chempi) + log(gas) + log(rtwex) +
                                   befile6 + affile6 + afdec6), barium)
ftest_manual_res = ftest(reg_manual_r.model, reg_manual_ur.model)

fstat_manual = ftest_manual_res.fstat[2]
fpval_manual = ftest_manual_res.pval[2]
println("fstat_manual = $fstat_manual\n")
println("fpval_manual = $fpval_manual")
```

Output of Script 12.3: `Example-12-4.jl`

```
fstat_manual = 5.12290705407486

fpval_manual = 0.00228980283295059
```

Another popular test is the Durbin-Watson test for AR(1) serial correlation. While the test statistic is pretty straightforward to compute, its distribution is non-standard and depends on the data. The package **`HypothesisTests`** provides the function **`DurbinWatsonTest`**, which calculates test statistic and p value. The test statistic ranges from 0 to 4, where 2 represents the case of no serial correlation. A value towards 0 indicates positive serial correlation, a value towards 4 negative serial correlation.

Script 12.4 (`Example-DWtest.jl`) repeats Example 12.2 but conducts DW tests instead of the t tests. The conclusions are the same: For the static model, no serial correlation can be rejected at a 1% level with a test statistic of $DW = 0.8027$, because the p value is very small. For the expectation augmented Phillips curve, the null hypothesis cannot be rejected on any reasonable significance level because the p value is 0.3567.

Script 12.4: `Example-DWtest.jl`

```
using WooldridgeDatasets, GLM, DataFrames, HypothesisTests
include("../03/getMats.jl")

phillips = DataFrame(wooldridge("phillips"))
yt96 = subset(phillips, :year => ByRow(<=(1996)))

# estimation of both Phillips curve models and
# extraction of regressor matrices and residuals:
reg_s = lm(@formula(inf ~ unem), yt96)
X_s = getMats(formula(reg_s), yt96)[2]
resid_s = residuals(reg_s)

yt96.inf_diff1 = vcat(missing, diff(yt96.inf))
yt96 = yt96[Not(1), :]
reg_ea = lm(@formula(inf_diff1 ~ unem), yt96)
X_ea = getMats(formula(reg_ea), yt96)[2]
resid_ea = residuals(reg_ea)

# DW tests:
DW_s = DurbinWatsonTest(X_s, resid_s)
DW_ea = DurbinWatsonTest(X_ea, resid_ea)
println("DW_s: \n$DW_s\n")
println("DW_ea: \n$DW_ea")
```

Output of Script 12.4: `Example-DWtest.jl`

```
DW_s:
Durbin-Watson autocorrelation test
----------------------------------
Population details:
    parameter of interest:   sample autocorrelation parameter
    value under h_0:         "0"
    point estimate:          0.59865

Test summary:
    outcome with 95% confidence: reject h_0
    two-sided p-value:           <1e-05

Details:
    number of observations:     49
    DW statistic:               0.8027


DW_ea:
Durbin-Watson autocorrelation test
----------------------------------
Population details:
    parameter of interest:   sample autocorrelation parameter
    value under h_0:         "0"
    point estimate:          0.115176

Test summary:
    outcome with 95% confidence: fail to reject h_0
    two-sided p-value:           0.3567

Details:
    number of observations:     48
    DW statistic:               1.76965
```

12.2. FGLS Estimation

In this subsection we demonstrate the Cochrane-Orcutt estimator to do the FGLS estimation. Since we are not aware of an available implementation in *Julia* with this functionality, we switch to *Python*'s **`statsmodels`** module in this case. There is a simple way with the command **`GLSAR`**. It expects matrices of dependent and independent variables and reports the Cochrane-Orcutt estimator as demonstrated in Example 12.5.

Wooldridge, Example 12.5: Cochrane-Orcutt Estimation

We once again use the monthly data set `BARIUM` and the same model as before. Script 12.5 (`Example-12-5.jl`) estimates the model with OLS and then calls **`GLSAR`**. As expected, the results are very close to the Prais-Winsten estimates reported by Wooldridge (2019).

Script 12.5: Example-12-5.jl

```
using PyCall, WooldridgeDatasets, GLM, DataFrames
# install Python's statsmodels with: using Conda; Conda.add("statsmodels")
sm = pyimport("statsmodels.api")
include("../03/getMats.jl")

barium = DataFrame(wooldridge("barium"))

# definition of model and hypotheses:
f = @formula(log(chnimp) ~ 1 + log(chempi) + log(gas) + log(rtwex) +
                           befile6 + affile6 + afdec6)
xy = getMats(f, barium)
y = xy[1]
X = xy[2]

# perform the Cochrane-Orcutt estimation (iterative procedure):
reg = sm.GLSAR(y, X)
CORC_results = reg.iterative_fit(maxiter=100)

reg_rho = reg.rho
table = DataFrame(
    coefnames=["Intercept", "log(chempi)", "log(gas)", "log(rtwex)",
        "befile6", "affile6", "afdec6"],
    b_CORC=CORC_results.params,
    se_CORC=round.(CORC_results.bse, digits=5))

println("reg_rho = $reg_rho\n")
println("table: \n$table")
```

Output of Script 12.5: Example-12-5.jl

```
reg_rho = [0.29585312847400064]

table:
7×3 DataFrame
 Row | coefnames    b_CORC       se_CORC
     | String       Float64      Float64
     ---------------------------------------------
   1 | Intercept    -37.513      23.239
   2 | log(chempi)    2.94545     0.6477
   3 | log(gas)       1.06332     0.99156
   4 | log(rtwex)     1.1384      0.51491
   5 | befile6       -0.0173144   0.32139
   6 | affile6       -0.0331082   0.32381
   7 | afdec6        -0.577328    0.34407
```

12.3. Serial Correlation-Robust Inference with OLS

Unbiasedness and consistency of OLS are not affected by heteroscedasticity or serial correlation, but the standard errors are. Similar to the heteroscedasticity-robust standard errors discussed in Section 8.1, we can use a formula for the variance-covariance matrix, often referred to as Newey-West standard errors. Wooldridge (2019, Section 12.5) shows how to calculate these standard errors and we implement it in Script 12.6 (`calc-hac-se.jl`). Script 12.7 (`Example-12-1.jl`) compares usual and Newey-West standard errors with an example.

Script 12.6: `calc-hac-se.jl`

```
using LinearAlgebra

# for details, see Equations 12.41 - 12.43 in Wooldridge (2019)
function calc_hac_se(reg, g)
    n = nobs(reg)
    X = reg.mm.m
    n = size(X, 1)
    K = size(X, 2)
    u = residuals(reg)
    ser = sqrt(sum(u .^ 2) / (n - K))
    se_ols = coeftable(reg).cols[2]
    se_hac = zeros(K)

    for k in 1:K
        yk = X[:, k]
        Xk = X[:, (1:K).!=k]
        bk = inv(transpose(Xk) * Xk) * transpose(Xk) * yk
        rk = yk .- Xk * bk
        ak = rk .* u
        vk = sum(ak .^ 2)
        for h in 1:g
            sum_h = 2 * (1 - h / (g + 1)) * sum(ak[(h+1):n] .* ak[1:(n-h)])
            vk = vk + sum_h
        end

        se_hac[k] = (se_ols[k] / ser)^2 * sqrt(vk)
    end
    return se_hac
end
```

Wooldridge, Example 12.1: The Puerto Rican Minimum Wage

Script 12.7 (`Example-12-1.jl`) estimates a model for the employment rate depending on the minimum wage as well as the GNP in Puerto Rico and the US. After the model has been fitted by OLS, we call the function **`calc_hac_se`** on the regression object. With $g = 2$ we get the results for the HAC variance-covariance formula reported in Wooldridge (2019). Both results imply a significantly negative relation between the minimum wage and employment.

Script 12.7: `Example-12-1.jl`

```
using WooldridgeDatasets, GLM, DataFrames
include("calc-hac-se.jl")

prminwge = DataFrame(wooldridge("prminwge"))
prminwge.time = prminwge.year .- 1949

# OLS with regular SE:
reg = lm(@formula(log(prepop) ~ log(mincov) + log(prgnp) +
                                log(usgnp) + time), prminwge)

# OLS with HAC SE:
hac_se = calc_hac_se(reg, 2)

# print different SEs:
table = DataFrame(coefficients=coeftable(reg).rownms,
    b=round.(coef(reg), digits=5),
    se_default=round.(coeftable(reg).cols[2], digits=5),
    hac_se=round.(hac_se, digits=5))
println("table: \n$table")
```

Output of Script 12.7: `Example-12-1.jl`

```
table:
5×4 DataFrame
 Row │ coefficients  b         se_default  hac_se
     │ String        Float64   Float64     Float64
─────┼─────────────────────────────────────────
   1 │ (Intercept)   -6.66344     1.25783  1.43179
   2 │ log(mincov)   -0.21226     0.04015  0.0426
   3 │ log(prgnp)     0.28524     0.08049  0.09285
   4 │ log(usgnp)     0.48605     0.22198  0.2601
   5 │ time          -0.02666     0.00463  0.00536
```

12.4. Autoregressive Conditional Heteroscedasticity

In time series, especially in financial data, a specific form of heteroscedasticity is often present. Autoregressive conditional heteroscedasticity (ARCH) and related models try to capture these effects.

Consider a basic linear time series equation

$$y_t = \beta_0 + \beta_1 x_{t1} + \beta_2 x_{t2} + \cdots + \beta_k x_{tk} + u_t. \tag{12.2}$$

The error term u follows an ARCH process if

$$\mathrm{E}(u_t^2|u_{t-1}, u_{t-2}, ...) = \alpha_0 + \alpha_1 u_{t-1}^2. \tag{12.3}$$

As the equation suggests, we can estimate α_0 and α_1 by an OLS regression of the residuals $\hat{u}_t^2$ on $\hat{u}_{t-1}^2$.

Wooldridge, Example 12.9: ARCH in Stock Returns

Script 12.8 (`Example-12-9.jl`) estimates a simple AR(1) model for weekly NYSE stock returns, already studied in Example 11.4. After the squared residuals are obtained, they are regressed on their lagged values. The coefficients from this regression are estimates for α_0 and α_1.

Script 12.8: **Example-12-9.jl**

```
using WooldridgeDatasets, GLM, DataFrames

nyse = DataFrame(wooldridge("nyse"))
nyse.ret = nyse.return
nyse.ret_lag1 = lag(nyse.ret, 1)
nyse = nyse[Not(1, 2), :]

# linear regression of model:
reg = lm(@formula(ret ~ ret_lag1), nyse)

# squared residuals:
nyse.resid_sq = residuals(reg) .^ 2
nyse.resid_sq_lag1 = lag(nyse.resid_sq, 1)

# model for squared residuals:
ARCHreg = lm(@formula(resid_sq ~ resid_sq_lag1), nyse)
table_ARCHreg = coeftable(ARCHreg)
println("table_ARCHreg: \n$table_ARCHreg")
```

Output of Script 12.8: **Example-12-9.jl**

```
table_ARCHreg:

                  Coef.  Std. Error     t  Pr(>|t|)  Lower 95%  Upper 95%

(Intercept)    2.94743    0.440234   6.70    <1e-10   2.08306    3.8118
resid_sq_lag1  0.337062   0.0359468  9.38    <1e-19   0.266483   0.407641
```

As a second example, let us reconsider the daily stock returns from Script 11.2 (`Example-EffMkts.jl`). We again download the daily Apple stock prices from Yahoo Finance and calculate their returns. Figure 11.1 on page 200 plots them. They show a very typical pattern for an ARCH-type of model: there are periods with high (such as fall 2008) and other periods with low volatility (fall 2010). In Script 12.9 (`Example-ARCH.jl`), we estimate an AR(1) process for the squared residuals. The t statistic is larger than 8, so there is very strong evidence for autoregressive conditional heteroscedasticity.

Script 12.9: `Example-ARCH.jl`

```
using DataFrames, GLM, Dates, MarketData

# download data for "AAPL" (= Apple) and define start and end:
ticker = "AAPL"
start_date = DateTime(2007, 12, 31)
end_date = DateTime(2017, 01, 01)

# import data as DataFrame:
AAPL_data = DataFrame(yahoo(ticker,
    YahooOpt(period1=start_date, period2=end_date)))

# calculate return as the difference of logged prices:
AAPL_data.ret = vcat(missing, diff(log.(AAPL_data.AdjClose)))
AAPL_data.ret_lag1 = lag(AAPL_data.ret, 1)
AAPL_data = AAPL_data[Not(1, 2), :]

# AR(1) model for returns:
reg = lm(@formula(ret ~ ret_lag1), AAPL_data)

# squared residuals:
AAPL_data.resid_sq = residuals(reg) .^ 2
AAPL_data.resid_sq_lag1 = lag(AAPL_data.resid_sq, 1)

# model for squared residuals:
ARCHreg = lm(@formula(resid_sq ~ resid_sq_lag1), AAPL_data)
table_ARCHreg = coeftable(ARCHreg)
println("table_ARCHreg: \n$table_ARCHreg")
```

Output of Script 12.9: `Example-ARCH.jl`

```
table_ARCHreg:

                   Coef.  Std. Error      t  Pr(>|t|)    Lower 95%    Upper 95%

(Intercept)    0.000345268  2.84054e-5  12.16    <1e-32  0.000289565  0.000400971
resid_sq_lag1  0.172245     0.0207069    8.32    <1e-15  0.131639     0.212852
```

Part III.

Advanced Topics

13. Pooling Cross Sections Across Time: Simple Panel Data Methods

Pooled cross sections consist of random samples from the same population at different points in time. Section 13.1 introduces this type of data set and how to use it for estimating changes over time. Section 13.2 covers difference-in-differences estimators, an important application of pooled cross sections for identifying causal effects.

Panel data resemble pooled cross-sectional data in that we have observations at different points in time. The key difference is that we observe the *same* cross-sectional units, for example individuals or firms. Panel data methods require the data to be organized in a systematic way, as discussed in Section 14.1. Section 13.4 introduces the first panel data method, first differenced estimation.

13.1. Pooled Cross Sections

If we have random samples at different points in time, this does not only increase the overall sample size and thereby the statistical precision of our analyses. It also allows to study changes over time and shed additional light on relationships between variables.

Wooldridge, Example 13.2: Changes to the Return to Education and the Gender Wage Gap

The data set `cps78_85` includes two pooled cross sections for the years 1978 and 1985. The dummy variable `y85` is equal to one for observations in 1985 and to zero for 1978. We estimate a model for the log wage `lwage` of the form

$$\begin{aligned}\texttt{lwage} = \beta_0 + \delta_0\texttt{y85} + \beta_1\texttt{educ} + \delta_1(\texttt{y85}\cdot\texttt{educ}) + \beta_2\texttt{exper} + \beta_3\tfrac{\texttt{exper}^2}{100}\\ + \beta_4\texttt{union} + \beta_5\texttt{female} + \delta_5(\texttt{y85}\cdot\texttt{female}) + u.\end{aligned}$$

Note that we divide $\texttt{exper}^2$ by 100 and thereby multiply β_3 by 100 compared to the results reported in Wooldridge (2019). The parameter β_1 measures the return to education in 1978 and δ_1 is the *difference* of the return to education in 1985 relative to 1978. Likewise, β_5 is the gender wage gap in 1978 and δ_5 is the change of the wage gap.
Script 13.1 (`Example-13-2.jl`) estimates the model. The return to education is estimated to have increased by $\hat{\delta}_1 = 0.0185$ and the gender wage gap decreased in absolute value from $\hat{\beta}_5 = -0.3167$ to $\hat{\beta}_5 + \hat{\delta}_5 = -0.2316$, even though this change is only marginally significant. The interpretation and implementation of interactions were covered in more detail in Section 6.1.6.

Script 13.1: `Example-13-2.jl`

```
using WooldridgeDatasets, GLM, DataFrames

cps78_85 = DataFrame(wooldridge("cps78_85"))

# OLS results including interaction terms:
reg = lm(@formula(lwage ~ y85 * (educ + female) + exper +
                         ((exper^2) / 100) + union), cps78_85)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Output of Script 13.1: `Example-13-2.jl`

```
table_reg:

                      Coef.  Std. Error      t  Pr(>|t|)      Lower 95%   Upper 95%

(Intercept)        0.458933   0.0934485    4.91    <1e-05   0.275571       0.642295
y85                0.117806   0.123782     0.95    0.3415  -0.125075       0.360687
educ               0.0747209  0.00667643  11.19    <1e-26   0.0616206      0.0878212
female            -0.316709   0.0366215   -8.65    <1e-16  -0.388566      -0.244851
exper              0.0295843  0.00356731   8.29    <1e-15   0.0225846      0.036584
exper ^ 2 / 100   -0.0399428  0.00775391  -5.15    <1e-06  -0.0551573     -0.0247283
union              0.202132   0.0302945    6.67    <1e-10   0.142689       0.261575
y85 & educ         0.0184605  0.00935417   1.97    0.0487   0.000106032    0.036815
y85 & female       0.085052   0.051309     1.66    0.0977  -0.0156251      0.185729
```

13.2. Difference-in-Differences

Wooldridge (2019, Section 13.2) discusses an important type of application for pooled cross sections. Difference-in-differences (DiD) estimators estimate the effect of a policy intervention (in the broadest sense) by comparing the change over time of an outcome of interest between an affected and an unaffected group of observations.

In a regression framework, we regress the outcome of interest on a dummy variable for the affected ("treatment") group, a dummy indicating observations after the treatment and an interaction term between both. The coefficient of this interaction term can then be a good estimator for the effect of interest, controlling for initial differences between the groups and contemporaneous changes over time.

Wooldridge, Example 13.3: Effect of a Garbage Incinerator's Location on Housing Prices

We are interested in whether and how much the construction of a new garbage incinerator affected the value of nearby houses. Script 13.2 (`Example-13-3-1.jl`) uses the data set `KIELMC`. We first estimate separate models for 1978 (before there were any rumors about the new incinerator) and 1981 (when the construction began). In 1981, the houses close to the construction site were cheaper by an average of $\$30,688.27$. But this was not only due to the new incinerator since even in 1978, nearby houses were cheaper by an average of $\$18,824.37$. The difference of these differences $\hat{\delta} = \$30,688.27 - \$18,824.37 = \$11,863.90$ is the DiD estimator and is arguably a better indicator of the actual effect.
The DiD estimator can be obtained more conveniently using a joint regression model with the interaction term as described above. The estimator $\hat{\delta} = \$11,863.90$ can be directly seen as the coefficient of the

interaction term. Conveniently, standard regression tables include t tests of the hypothesis that the actual effect is equal to zero. For a one-sided test, the p value is $\frac{1}{2} \cdot 0.113 = 0.056$ (not reported in the output), so there is some statistical evidence of a negative impact.
The DiD estimator can be improved. A logarithmic specification is more plausible since it implies a constant *percentage* effect on the house values. We can also add additional regressors to control for incidental changes in the composition of the houses traded. Script 13.3 (Example-13-3-2.jl) implements both improvements. The model including features of the houses implies an estimated decrease in the house values of about 13.2%. This effect is also significantly different from zero.

Script 13.2: Example-13-3-1.jl

```
using WooldridgeDatasets, GLM, DataFrames, RegressionTables

kielmc = DataFrame(wooldridge("kielmc"))
kielmc.is1981 = kielmc.year .== 1981

# separate regressions for 1978 and 1981:
y78 = subset(kielmc, :year => ByRow(==(1978)))
reg78 = lm(@formula(rprice ~ nearinc), y78)

y81 = subset(kielmc, :year => ByRow(==(1981)))
reg81 = lm(@formula(rprice ~ nearinc), y81)

# joint regression including an interaction term:
reg_joint = lm(@formula(rprice ~ nearinc * is1981), kielmc)

# print results with RegressionTables:
regtable(reg78, reg81, reg_joint)
```

Output of Script 13.2: Example-13-3-1.jl

```
-----------------------------------------------------------------
                                          rprice
                    ---------------------------------------------
                              (1)             (2)             (3)
-----------------------------------------------------------------
(Intercept)          82517.228***   101307.514***    82517.228***
                       (2653.790)      (3093.027)      (2726.910)
nearinc             -18824.370***   -30688.274***   -18824.370***
                       (4744.594)      (5827.709)      (4875.322)
is1981                                               18790.286***
                                                       (4050.065)
nearinc & is1981                                       -11863.903
                                                       (7456.646)
-----------------------------------------------------------------
Estimator                     OLS             OLS             OLS
-----------------------------------------------------------------
N                             179             142             321
R2                          0.082           0.165           0.174
-----------------------------------------------------------------
```

Script 13.3: Example-13-3-2.jl

```
using WooldridgeDatasets, GLM, DataFrames

kielmc = DataFrame(wooldridge("kielmc"))
kielmc.is1981 = kielmc.year .== 1981

# difference in difference (DiD):
reg_did = lm(@formula(log(rprice) ~ nearinc * is1981), kielmc)
table_did = coeftable(reg_did)
println("table_did: \n$table_did\n")

# DiD with control variables:
reg_didC = lm(@formula(log(rprice) ~ nearinc * is1981 + age + (age^2) +
                                     log(intst) + log(land) + log(area) +
                                     rooms + baths), kielmc)
table_didC = coeftable(reg_didC)
println("table_didC: \n$table_didC")
```

Output of Script 13.3: Example-13-3-2.jl

```
table_did:

                     Coef.  Std. Error       t  Pr(>|t|)  Lower 95%  Upper 95%

(Intercept)       11.2854     0.0305145  369.84    <1e-99  11.2254    11.3455
nearinc           -0.339923   0.0545555   -6.23    <1e-08  -0.44726   -0.232587
is1981             0.193094   0.0453207    4.26    <1e-04   0.103926   0.282261
nearinc & is1981  -0.062649   0.0834408   -0.75    0.4533  -0.226817   0.101519


table_didC:

                       Coef.  Std. Error      t  Pr(>|t|)    Lower 95%     Upper 95%

(Intercept)         7.6517     0.415884    18.40    <1e-50   6.83339      8.47001
nearinc             0.0322389  0.0474875    0.68    0.4977  -0.0611997    0.125678
is1981              0.162074   0.0284999    5.69    <1e-07   0.105997     0.218152
age               -0.00835901  0.00141115  -5.92    <1e-08  -0.0111356   -0.00558237
age ^ 2             3.76343e-5 8.66848e-6   4.34    <1e-04   2.05778e-5   5.46908e-5
log(intst)        -0.0614386   0.0315075   -1.95    0.0521  -0.123434     0.000557068
log(land)           0.0998402  0.0244909    4.08    <1e-04   0.0516508    0.14803
log(area)           0.350774   0.0514866    6.81    <1e-10   0.249467     0.452081
rooms               0.0473335  0.0173274    2.73    0.0067   0.0132392    0.0814278
baths               0.0942765  0.0277257    3.40    0.0008   0.0397221    0.148831
nearinc & is1981  -0.131514    0.0519713   -2.53    0.0119  -0.233775    -0.0292527
```

13.3. Organizing Panel Data

A panel data set includes several observations at different points in time t for the same (or at least an overlapping) set of cross-sectional units i. A simple "pooled" regression model could look like

$$y_{it} = \beta_0 + \beta_1 x_{it1} + \beta_2 x_{it2} + \cdots + \beta_k x_{itk} + v_{it}; \qquad t = 1, \ldots, T; \qquad i = 1, \ldots, n, \tag{13.1}$$

where the double subscript now indicates values for individual (or other cross-sectional unit) i at time t. We could estimate this model by OLS, essentially ignoring the panel structure. But at least the assumption that the error terms are unrelated is very hard to justify since they contain unobserved individual traits that are likely to be constant or at least correlated over time. Therefore, we need specific methods for panel data.

For the calculations used by panel data methods, we have to make sure that the data set is systematically organized and the estimation routines understand its structure. Usually, a panel data set comes in a "long" form where each row of data corresponds to one combination of i and t. We have to define which observations belong together by introducing a variable for the cross-sectional units i and preferably also the time index t. In Script 13.4 (`Example-FD.jl`), for example, we use the variables **`id`** to identify cross-sectional units and **`year`** as the time variable.

13.4. First Differenced Estimator

Wooldridge (2019, Sections 13.3 – 13.5) discusses basic unobserved effects models and their estimation by first-differencing (FD). Consider the model

$$y_{it} = \beta_0 + \beta_1 x_{it1} + \cdots + \beta_k x_{itk} + a_i + u_{it}; \qquad t = 1, \ldots, T; \qquad i = 1, \ldots, n, \tag{13.2}$$

which differs from Equation 13.1 in that it explicitly involves an unobserved effect a_i that is constant over time (since it has no t subscript). If it is correlated with one or more of the regressors $x_{it1}, \ldots, x_{itk}$, we cannot simply ignore a_i, leave it in the composite error term $v_{it} = a_i + u_{it}$ and estimate the equation by OLS. The error term v_{it} would be related to the regressors, violating assumption MLR.4 (and MLR.4') and creating biases and inconsistencies. Note that this problem is not unique to panel data, but possible solutions are.

The first differenced (FD) estimator is based on the first difference of the whole equation:

$$\begin{aligned} \Delta y_{it} &\equiv y_{it} - y_{it-1} \\ &= \beta_1 \Delta x_{it1} + \cdots + \beta_k \Delta x_{itk} + \Delta u_{it}; \qquad t = 2, \ldots, T; \qquad i = 1, \ldots, n. \end{aligned} \tag{13.3}$$

Note that we cannot evaluate this equation for the first observation $t = 1$ for any i since the lagged values are unknown for them. The trick is that a_i drops out of the equation by differencing since it does not change over time. No matter how badly it is correlated with the regressors, it cannot hurt the estimation anymore. This estimating equation is then analyzed by OLS. We simply regress the differenced dependent variable Δy_{it} on the differenced independent variables $\Delta x_{it1}, \ldots, \Delta x_{itk}$.

Script 13.4 (`Example-FD.jl`) opens the data set `CRIME2` already described above. We describe the data preparation required for the manual estimation. First, we need to sort the data by **`id`** and **`year`** to make sure the same temporal difference is calculated for each city. Before we can use the function **`diff`** to calculate first differences of the dependent variable crime rate (**`crmrte`**) and the independent variable unemployment rate (**`unem`**), we have to make sure that these calculations are performed per individual with **`grouped_df = groupby(crime2, :id)`**. With the following line of code, we calculate the differences for the variable **`crmrte`** and combine the results in a data frame:

```
combine(grouped_df, :crmrte => diff).crmrte_diff
```

The first five observations reveal that there is only one difference for each city, which makes sense, because there are only two years in the data set. For example the change of the crime rate for city 1 is $70.11729 - 74.65756 = -4.54027$ and the change of the unemployment rate for city 2 is $5.4 - 8.1 = -2.7$. The FD estimator can now be calculated by simply applying OLS to these differenced values. The observations for the first year with missing information are automatically dropped from the estimation sample. The results show a significantly positive relation between unemployment and crime. Script 13.5 (`Example-13-9.jl`) gives another example.

Script 13.4: Example-FD.jl

```
using WooldridgeDatasets, GLM, DataFrames

crime2 = DataFrame(wooldridge("crime2"))

# create an index in this balanced data set by combining two vectors:
id_tmp = 1:46
crime2.id = sort(vcat(id_tmp, id_tmp))

# sort data by id and year:
sort!(crime2, [:id, :year])

# manually calculate first differences per entity for crmrte and unem:
grouped_df = groupby(crime2, :id)
diff_df = DataFrame(id=id_tmp)
diff_df.crmrte_diff1 = combine(grouped_df, :crmrte => diff).crmrte_diff
diff_df.unem_diff1 = combine(grouped_df, :unem => diff).unem_diff

preview = diff_df[1:5, :]
println("preview: \n$preview\n")

# estimate FD model with OLS on differenced data:
reg_sm = lm(@formula(crmrte_diff1 ~ unem_diff1), diff_df)
table_sm = coeftable(reg_sm)
println("table_sm: \n$table_sm")
```

Output of Script 13.4: Example-FD.jl

```
preview:
5×3 DataFrame
 Row | id     crmrte_diff1  unem_diff1
     | Int64  Float64       Float64
     ---------------------------------------------
   1 |     1      -4.54027        -4.5
   2 |     2      -2.96265        -2.7
   3 |     3      -6.41637        -3.1
   4 |     4      -4.90154        -6.9
   5 |     5      -4.60899        -5.2

table_sm:

               Coef.  Std. Error     t  Pr(>|t|)  Lower 95%  Upper 95%

(Intercept)  15.4022    4.70212   3.28    0.0021   5.92571    24.8787
unem_diff1    2.218     0.877866  2.53    0.0152   0.448777    3.98722
```

Wooldridge, Example 13.9: County Crime Rates in North Carolina

Script 13.5 (Example-13-9.jl) analyzes the data CRIME4. We estimate the model in first differences using the same approach as in Script 13.4 (Example-FD.jl), but for more variables.
Note that in this specification, all variables are differenced, so they have the intuitive interpretation in the level equation. In the results reported by Wooldridge (2019), the year dummies are not differenced which only makes a difference for the interpretation of the year coefficients.

Script 13.5: Example-13-9.jl

```
using WooldridgeDatasets, GLM, DataFrames

crime4 = DataFrame(wooldridge("crime4"))
crime4.lcrmrte = log.(crime4.crmrte)

# sort data by county and year:
sort!(crime4, [:county, :year])

# manually calculate first differences for multiple variables:
vars_to_diff = ["lcrmrte", "d83", "d84", "d85", "d86", "d87",
    "lprbarr", "lprbconv", "lprbpris", "lavgsen", "lpolpc"]
grouped_df = groupby(crime4, :county)
diff_df = DataFrame()

for i in vars_to_diff
    tmp_diff_i = combine(grouped_df, Symbol(i) => diff)[:, 2]
    diff_df[!, i] = tmp_diff_i
end

# estimate FD model:
reg = lm(@formula(lcrmrte ~ d83 + d84 + d85 + d86 + d87 +
                            lprbarr + lprbconv + lprbpris +
                            lavgsen + lpolpc), diff_df)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Output of Script 13.5: Example-13-9.jl

```
table_reg:

                   Coef.  Std. Error       t  Pr(>|t|)   Lower 95%   Upper 95%

(Intercept)   0.00771336   0.0170579    0.45    0.6513  -0.0257961   0.0412229
d83          -0.0998658    0.0238953   -4.18    <1e-04  -0.146807   -0.0529246
d84          -0.147803     0.0412794   -3.58    0.0004  -0.228895   -0.0667117
d85          -0.152414     0.0584      -2.61    0.0093  -0.267139   -0.0376901
d86          -0.1249       0.0760042   -1.64    0.1009  -0.274207    0.024407
d87          -0.0840734    0.0940003   -0.89    0.3715  -0.268733    0.100586
lprbarr      -0.327494     0.0299801  -10.92    <1e-24  -0.386389   -0.268599
lprbconv     -0.238107     0.0182341  -13.06    <1e-33  -0.273927   -0.202286
lprbpris     -0.165046     0.025969    -6.36    <1e-09  -0.216061   -0.114031
lavgsen      -0.0217606    0.0220909   -0.99    0.3251  -0.0651573   0.0216361
lpolpc        0.398426     0.026882    14.82    <1e-41   0.345618    0.451235
```

14. Advanced Panel Data Methods

In this chapter, we look into additional panel data models and methods. We start with the widely used fixed effects (FE) estimator in Section 14.2, followed by random effects (RE) in Section 14.3. The dummy variable regression and correlated random effects approaches presented in Section 14.4 can be used as alternatives and generalizations of FE. We will come back to panel data in combination with instrumental variables in Section 15.6.

14.1. Getting Started with Panel Data

We will use the package **Econometrics**, which is a comprehensive collection of commands dealing with regression models including panel data estimators.[1] After installing it, the following line of code loads the package:

```
using Econometrics
```

The routines often require a data frame including two variables, which describe the individual and time dimensions.

14.2. Fixed Effects Estimation

We start from the same basic unobserved effects models as Equation 13.2. Instead of first differencing, we get rid of the unobserved individual effect a_i using the within transformation:

$$\begin{aligned} y_{it} &= \beta_0 + \beta_1 x_{it1} + \cdots + \beta_k x_{itk} + a_i + u_{it}; \qquad t = 1, \ldots, T; \qquad i = 1, \ldots, n, \\ \bar{y}_i &= \beta_0 + \beta_1 \bar{x}_{i1} + \cdots + \beta_k \bar{x}_{ik} + a_i + \bar{u}_i \\ \ddot{y}_{it} = y_{it} - \bar{y}_i &= \beta_1 \ddot{x}_{it1} + \cdots + \beta_k \ddot{x}_{itk} + \ddot{u}_{it}, \end{aligned} \tag{14.1}$$

where $\bar{y}_i$ is the average of y_{it} over time for cross-sectional unit i and for the other variables accordingly. The within transformation subtracts these individual averages from the respective observations y_{it}.

The fixed effects (FE) estimator simply estimates the demeaned Equation 14.1 using pooled OLS. Instead of applying the within transformation to all variables and running **`lm`**, we can simply use **`fit(EconometricModel, formula, dataframe)`** in the package **`Econometrics`**. The within transformation is considered by adding **`absorb(id)`** to the formula, where **`id`** is the variable identifying an individual. This has the additional advantage that the degrees of freedom are adjusted to the demeaning and the variance-covariance matrix and standard errors are adjusted accordingly. We will come back to different ways to get the same estimates in Section 14.4. This is shown in Script 14.1 (`Example-14-2.jl`).

[1] For more information about the package, see Calderón and Bayoán (2020) or `https://docs.juliahub.com/Econometrics/XQPLt/0.2.7/getting_started/`.

Wooldridge, Example 14.2: Has the Return to Education Changed over Time?

We estimate the change of the return to education over time using a fixed effects estimator. Script 14.1 (`Example-14-2.jl`) shows the implementation. The data set `WAGEPAN` is a balanced panel for $n = 545$ individuals over $T = 8$ years. The panel structure is described by the variables **`nr`** and **`year`** for individuals and years, respectively. We implement the demeaning on the individual level by including **`absorb(nr)`** in the formula. The formula is a little longer than necessary, because we could also use **`year => DummyCoding()`**, to get rid of all the year dummys (see Script 14.2 (`Example-14-4.jl`) for an alternative implementation). But with this specification we get the same results as Wooldridge (2019). Since **`educ`** does not change over time, we cannot estimate its overall impact in the estimation. However, we can interact it with time dummies to see how the impact changes over time.

Script 14.1: `Example-14-2.jl`

```
using WooldridgeDatasets, DataFrames, Econometrics

wagepan = DataFrame(wooldridge("wagepan"))

# FE model estimation:
reg = fit(EconometricModel,
    @formula(lwage ~ married + union +
                     d81 + d81 + d82 + d83 + d84 + d85 + d86 + d87 +
                     d81 & educ + d81 & educ + d82 & educ + d83 & educ +
                     d84 & educ + d85 & educ + d86 & educ + d87 & educ +
                     absorb(nr)),
    wagepan)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Output of Script 14.1: `Example-14-2.jl`

```
table_reg:

                      PE          SE        t-value  Pr > |t|         2.50%     97.50%

(Intercept)   1.36246     0.0162385  83.9031       <1e-99   1.33062      1.3943
married       0.0548205   0.0184126   2.97734      0.0029   0.018721     0.09092
union         0.0829785   0.0194461   4.2671       <1e-04   0.0448527    0.121104
d81          -0.0224158   0.145888   -0.15365      0.8779  -0.308443     0.263611
d82          -0.00576106  0.145856   -0.0394983    0.9685  -0.291724     0.280202
d83           0.0104297   0.145858    0.071506     0.9430  -0.275538     0.296397
d84           0.0843743   0.145852    0.578493     0.5630  -0.201581     0.37033
d85           0.0497253   0.14586     0.340911     0.7332  -0.236247     0.335697
d86           0.0656064   0.145892    0.449693     0.6530  -0.220427     0.35164
d87           0.0904448   0.145851    0.62012      0.5352  -0.195508     0.376398
d81 & educ    0.0115854   0.0122625   0.944788     0.3448  -0.0124562    0.0356271
d82 & educ    0.0147905   0.0122635   1.20605      0.2279  -0.00925326   0.0388342
d83 & educ    0.0171182   0.0122633   1.39589      0.1628  -0.00692509   0.0411615
d84 & educ    0.0165839   0.0122657   1.35206      0.1764  -0.00746404   0.0406319
d85 & educ    0.0237085   0.0122738   1.93163      0.0535  -0.000355375  0.0477725
d86 & educ    0.0274123   0.012274    2.23337      0.0256   0.00334806   0.0514765
d87 & educ    0.0304332   0.0122723   2.47982      0.0132   0.00637217   0.0544942
```

14.3. Random Effects Models

We again base our analysis on the basic unobserved effects model in Equation 13.2. The random effects (RE) model assumes that the unobserved effects a_i are independent of (or at least uncorrelated with) the regressors x_{itj} for all t and $j = 1, \ldots, k$. Therefore, our main motivation for using FD or FE disappears: OLS consistently estimates the model parameters under this additional assumption.

However, like the situation with heteroscedasticity (see Section 8.3) and autocorrelation (see Section 12.2), we can obtain more efficient estimates if we take into account the structure of the variances and covariances of the error term. Wooldridge (2019, Section 14.2) shows that the GLS transformation that takes care of their special structure implied by the RE model leads to a quasi-demeaned specification

$$\mathring{y}_{it} = y_{it} - \theta \bar{y}_i = \beta_0(1-\theta) + \beta_1 \mathring{x}_{it1} + \cdots + \beta_k \mathring{x}_{itk} + \mathring{v}_{it}, \tag{14.2}$$

where $\mathring{y}_{it}$ is similar to the demeaned $\ddot{y}_{it}$ from Equation 14.1 but subtracts only a fraction θ of the individual averages. The same holds for the regressors x_{itj} and the composite error term $v_{it} = a_i + u_{it}$.

The parameter $\theta = 1 - \sqrt{\frac{\sigma_u^2}{\sigma_u^2 + T\sigma_a^2}}$ depends on the variances of u_{it} and a_i and the length of the time series dimension T. It is unknown and has to be estimated. With the variable **id** describing the individual and **t** the time dimension in the data frame **sample**, we can estimate the RE model parameters in **Econometrics** using the following syntax:

```
fit(RandomEffectsEstimator, @formula(y ~ x1 + x2 + x3),
    sample,
    panel=:id,
    time=:t)
```

Unlike with FD and FE estimators, we can include variables in our model that are constant over time for each cross-sectional unit.

Wooldridge, Example 14.4: A Wage Equation Using Panel Data

The data set `WAGEPAN` was already used in Example 14.2. We get estimates using OLS, RE, and FE estimators in Script 14.2 (`Example-14-4.jl`). We use **lm**, **fit(RandomEffectsEstimator,...)** and **fit(EconometricModel,...)**, respectively.

Script 14.2: `Example-14-4.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Econometrics

wagepan = DataFrame(wooldridge("wagepan"))

# estimate different models:
reg_ols = lm(@formula(lwage ~ educ + black + hisp + exper + (exper^2) +
                              married + union + year),
    wagepan,
    contrasts=Dict(:year => DummyCoding()))

reg_re = fit(RandomEffectsEstimator,
    @formula(lwage ~ educ + black + hisp + exper + (exper^2) +
                     married + union + year),
    wagepan,
    panel=:nr,
    time=:year,
    contrasts=Dict(:year => DummyCoding()))
```

```
reg_fe = fit(EconometricModel,
    @formula(lwage ~ (exper^2) + married + union + year + absorb(nr)),
    wagepan,
    contrasts=Dict(:year => DummyCoding()))

# print results:
table_ols = coeftable(reg_ols)
println("table_ols: \n$table_ols\n")

table_re = coeftable(reg_re)
println("table_re: \n$table_re\n")

table_fe = coeftable(reg_fe)
println("table_fe: \n$table_fe")
```

Output of Script 14.2: `Example-14-4.jl`

```
table_ols:

                    Coef.   Std. Error      t  Pr(>|t|)    Lower 95%     Upper 95%

(Intercept)   0.0920558  0.0782701     1.18    0.2396  -0.0613935    0.245505
educ          0.0913498  0.00523738   17.44    <1e-65   0.0810819    0.101618
black        -0.139234   0.0235796    -5.90    <1e-08  -0.185462    -0.0930062
hisp          0.0160195  0.0207971     0.77    0.4412  -0.0247535    0.0567925
exper         0.0672345  0.0136948     4.91    <1e-06   0.0403856    0.0940834
exper ^ 2    -0.0024117  0.000819955  -2.94    0.0033  -0.00401923  -0.000804174
married       0.108253   0.0156894     6.90    <1e-11   0.0774937    0.139012
union         0.182461   0.0171568    10.63    <1e-25   0.148825     0.216097
year: 1981    0.05832    0.0303536     1.92    0.0548  -0.00118862   0.117829
year: 1982    0.0627744  0.0332141     1.89    0.0588  -0.0023421    0.127891
year: 1983    0.0620117  0.0366601     1.69    0.0908  -0.00986081   0.133884
year: 1984    0.0904672  0.0400907     2.26    0.0241   0.011869     0.169065
year: 1985    0.109246   0.0433525     2.52    0.0118   0.0242533    0.194239
year: 1986    0.14196    0.046423      3.06    0.0022   0.0509469    0.232972
year: 1987    0.173833   0.049433      3.52    0.0004   0.0769194    0.270747


table_re:

                     PE           SE       t-value  Pr > |t|        2.50%       97.50%

(Intercept)   0.0236081   0.150572      0.156789    0.8754  -0.27159      0.318806
educ          0.0918757   0.0106527     8.62462     <1e-17   0.070991     0.112761
black        -0.139376    0.0476917    -2.92245     0.0035  -0.232877    -0.0458764
hisp          0.0217296   0.0425783     0.510344    0.6098  -0.0617456    0.105205
exper         0.105743    0.015362      6.88341     <1e-11   0.0756255    0.13586
exper ^ 2    -0.00472328  0.000689539  -6.8499      <1e-11  -0.00607513  -0.00337143
married       0.0640076   0.0167738     3.81593     0.0001   0.0311224    0.0968928
union         0.106168    0.0178536     5.9466      <1e-08   0.0711659    0.14117
year: 1981    0.0404669   0.0246932     1.63879     0.1013  -0.00794436   0.0888781
year: 1982    0.03093     0.0323331     0.956604    0.3388  -0.0324594    0.0943194
year: 1983    0.0202923   0.0415657     0.488197    0.6254  -0.0611978    0.101782
year: 1984    0.0431321   0.0512922     0.840909    0.4004  -0.0574267    0.143691
year: 1985    0.0578306   0.0612001     0.944942    0.3447  -0.0621529    0.177814
year: 1986    0.0919627   0.071189      1.29181     0.1965  -0.0476041    0.23153
year: 1987    0.134941    0.0812653     1.6605      0.0969  -0.0243808    0.294262
```

```
table_fe:

                 PE          SE        t-value  Pr > |t|       2.50%       97.50%

(Intercept)   1.42602    0.0183415    77.7484     <1e-99   1.39006      1.46198
exper ^ 2    -0.0051855  0.000704437  -7.3612     <1e-12  -0.00656661  -0.00380439
married       0.0466804  0.0183104     2.54939    0.0108   0.0107811    0.0825796
union         0.0800019  0.0193103     4.14296    <1e-04   0.0421423    0.117861
year: 1981    0.151191   0.0219489     6.88832    <1e-11   0.108158     0.194224
year: 1982    0.252971   0.0244185    10.3598     <1e-24   0.205096     0.300845
year: 1983    0.354444   0.0292419    12.1211     <1e-32   0.297113     0.411775
year: 1984    0.490115   0.0362266    13.5291     <1e-40   0.419089     0.56114
year: 1985    0.617482   0.0452435    13.648      <1e-40   0.528778     0.706186
year: 1986    0.765497   0.0561277    13.6385     <1e-40   0.655453     0.87554
year: 1987    0.925025   0.0687731    13.4504     <1e-39   0.790189     1.05986
```

The RE estimator needs stronger assumptions to be consistent than the FE estimator. On the other hand, it is more efficient if these assumptions hold and we can include time constant regressors. A widely used test of this additional assumption is the Hausman test, which is based on the comparison between the FE and RE parameter estimates.

14.4. Dummy Variable Regression and Correlated Random Effects

It turns out that we can get the FE parameter estimates in two other ways than the within transformation we used in Section 14.2. The dummy variable regression uses OLS on the original variables in Equation 13.2 instead of the transformed ones. But it adds $n-1$ dummy variables (or n dummies and removes the constant), one for each cross-sectional unit $i = 1, \ldots, n$. The simplest (although not the computationally most efficient) way to implement this in *Julia* is to use the cross-sectional index as another categorical variable.

The third way to get the same results is the correlated random effects (CRE) approach. Instead of assuming that the individual effects a_i are independent of the regressors x_{itj}, we assume that they only depend on the averages over time $\bar{x}_{ij} = \frac{1}{T}\sum_{t=1}^{T} x_{itj}$:

$$a_i = \gamma_0 + \gamma_1 \bar{x}_{i1} + \cdots + \gamma_k \bar{x}_{ik} + r_i \tag{14.3}$$

$$\begin{aligned} y_{it} &= \beta_0 + \beta_1 x_{it1} + \cdots + \beta_k x_{itk} + a_i + u_{it} \\ &= \beta_0 + \gamma_0 + \beta_1 x_{it1} + \cdots + \beta_k x_{itk} + \gamma_1 \bar{x}_{i1} + \cdots + \gamma_k \bar{x}_{ik} + r_i + u_{it}. \end{aligned} \tag{14.4}$$

If r_i is uncorrelated with the regressors, we can consistently estimate the parameters of this model using the RE estimator. In addition to the original regressors, we include their averages over time.

Script 14.3 (`Example-Dummy-CRE.jl`) uses `WAGEPAN` again. We estimate the FE parameters using the within transformation (**`reg_we`**), the dummy variable approach (**`reg_dum`**), and the CRE approach (**`reg_cre`**). We also estimate the RE version of this model (**`reg_re`**). The results confirm that the first three methods deliver exactly the same parameter estimates, while the RE estimates differ.

Script 14.3: `Example-Dummy-CRE.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Econometrics, Statistics

wagepan = DataFrame(wooldridge("wagepan"))

# include group specific means:
grouped_means = combine(groupby(wagepan, :nr), [:married, :union] .=> mean)
wagepan = innerjoin(grouped_means, wagepan, on=:nr)

# estimate FE parameters in 3 different ways:
reg_we = fit(EconometricModel,
    @formula(lwage ~ married + union + absorb(nr) +
                     d81 + d81 + d82 + d83 + d84 + d85 + d86 + d87 +
                     d81 & educ + d81 & educ + d82 & educ + d83 & educ +
                     d84 & educ + d85 & educ + d86 & educ + d87 & educ),
    wagepan)

reg_dum = lm(@formula(lwage ~ married + union + year * educ + nr),
    wagepan,
    contrasts=Dict(:year => DummyCoding(), :nr => DummyCoding()))


reg_cre = fit(RandomEffectsEstimator,
    @formula(lwage ~ married + union + year * educ +
                     married_mean + union_mean),
    wagepan,
    panel=:nr,
    time=:year,
    contrasts=Dict(:year => DummyCoding()))

# compare to RE estimates:
reg_re = fit(RandomEffectsEstimator,
    @formula(lwage ~ married + union + year * educ),
    wagepan,
    panel=:nr,
    time=:year,
    contrasts=Dict(:year => DummyCoding()))

# print results for married and union:
table = DataFrame(coef_names=["married", "union"],
    b_we=round.(coef(reg_we)[[2, 3]], digits=5),
    b_dum=round.(coef(reg_dum)[[2, 3]], digits=5),
    b_cre=round.(coef(reg_cre)[[2, 3]], digits=5),
    b_re=round.(coef(reg_re)[[2, 3]], digits=5))
println("table:\n $table")
```

Output of Script 14.3: `Example-Dummy-CRE.jl`

```
table:
 2×5 DataFrame
 Row | coef_names  b_we     b_dum    b_cre    b_re
     | String      Float64  Float64  Float64  Float64
     ---------------------------------------------
   1 | married     0.05482  0.05482  0.05482  0.07293
   2 | union       0.08298  0.08298  0.08298  0.10267
```

An advantage of the CRE approach is that we can add time-constant regressors to the model. Since we cannot control for average values $\bar{x}_{ij}$ for these variables, they have to be uncorrelated with a_i for consistent estimation of *their* coefficients. For the other coefficients of the time-varying variables, we still don't need these additional RE assumptions.

Script 14.4 (`Example-CRE.jl`) estimates another version of the wage equation using the CRE approach. The variables **`married`** and **`union`** vary over time, so we can control for their between effects. The variables **`educ`**, **`black`**, and **`hisp`** do not vary. For a causal interpretation of *their* coefficients, we have to rely on uncorrelatedness with a_i. Given a_i includes intelligence and other labor market success factors, this uncorrelatedness is more plausible for some variables (like gender or race) than for other variables (like education).

Script 14.4: `Example-CRE.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Econometrics, Statistics

wagepan = DataFrame(wooldridge("wagepan"))

# include group specific means:
grouped_means = combine(groupby(wagepan, :nr), [:married, :union] .=> mean)
wagepan = innerjoin(grouped_means, wagepan, on=:nr)

# estimate CRE:
reg_CRE = fit(RandomEffectsEstimator,
    @formula(lwage ~ married + union + educ + black +
                     hisp + married_mean + union_mean),
    wagepan,
    panel=:nr,
    time=:year)
table_reg = coeftable(reg_CRE)
println("table_reg: \n$table_reg")
```

Output of Script 14.4: `Example-CRE.jl`

```
table_reg:

                   PE          SE         t-value  Pr > |t|       2.50%       97.50%

(Intercept)     0.632563   0.108154     5.8487      <1e-08    0.420525    0.844601
married         0.241684   0.0176735   13.675       <1e-40    0.207036    0.276333
union           0.0700438  0.020724     3.37985     0.0007    0.0294143   0.110673
educ            0.0760374  0.00877868   8.6616      <1e-17    0.0588267   0.0932481
black          -0.129516   0.0488981   -2.6487      0.0081   -0.225381   -0.0336511
hisp            0.01167    0.0428188    0.272543    0.7852   -0.0722767   0.0956167
married_mean   -0.0797386  0.0442674   -1.80129     0.0717   -0.166525    0.00704806
union_mean      0.191855   0.0506522    3.78769     0.0002    0.0925505   0.291159
```

15. Instrumental Variables Estimation and Two Stage Least Squares

Instrumental variables are potentially powerful tools for the identification and estimation of causal effects. We start the discussion in Section 15.1 with the simplest case of one endogenous regressor and one instrumental variable. Section 15.2 shows how to implement models with additional exogenous regressors. In Section 15.3, we will introduce two stage least squares which efficiently deals with several endogenous variables and several instruments.

Tests of the exogeneity of the regressors and instruments are presented in Sections 15.4 and 15.5, respectively. Finally, Section 15.6 shows how to conveniently combine panel data estimators with instrumental variables.

15.1. Instrumental Variables in Simple Regression Models

We start the discussion of instrumental variables (IV) regression with the most straightforward case of only one regressor and only one instrumental variable. Consider the simple linear regression model for cross-sectional data

$$y = \beta_0 + \beta_1 x + u. \tag{15.1}$$

The OLS estimator for the slope parameter is $\hat{\beta}_1^{\text{OLS}} = \frac{\text{Cov}(x,y)}{\text{Var}(x)}$, see Equation 2.3. Suppose the regressor x is correlated with the error term u, so OLS parameter estimators will be biased and inconsistent.

If we have a valid instrumental variable z, we can consistently estimate β_1 using the IV estimator

$$\hat{\beta}_1^{\text{IV}} = \frac{\text{Cov}(z,y)}{\text{Cov}(z,x)}. \tag{15.2}$$

A valid instrument is correlated with the regressor x ("relevant"), so the denominator of Equation 15.2 is nonzero. It is also uncorrelated with the error term u ("exogenous"). Wooldridge (2019, Section 15.1) provides more discussion and examples.

To implement IV regression in *Julia*, the package **`Econometrics`** provides the functionality including the convenient formula syntax we know from **`GLM`**.

In the formula specification, the endogenous regressor(s) **`x_end`** and instruments **`z`** are provided in the following way:

```
y ~  (x_end ~ z)
```

Wooldridge, Example 15.1: Return to Education for Married Women

Script 15.1 (`Example-15-1.jl`) uses data from `MROZ`. We only analyze women with non-missing wage, so we use the function **ismissing** to extract them. We want to estimate the return to education (**educ**) for these women. As an instrumental variable for education, we use the education of her father (**fatheduc**).

First, we calculate the OLS and IV slope parameters according to Equations 2.3 and 15.2. Then, the full OLS and IV estimates are calculated using the boxed routines **lm** and **fit(EconometricModel,...)**, respectively. Not surprisingly, the slope parameters match the manual results.

Script 15.1: `Example-15-1.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Econometrics, Statistics

mroz_wm = DataFrame(wooldridge("mroz"))

# restrict to non-missing wage observations:
mroz = mroz_wm[.!ismissing.(mroz_wm.wage), :]

# OLS slope parameter manually:
cov_yz = cov(mroz.lwage, mroz.fatheduc)
cov_xy = cov(mroz.educ, mroz.lwage)
cov_xz = cov(mroz.educ, mroz.fatheduc)
var_x = var(mroz.educ)
x_bar = mean(mroz.educ)
y_bar = mean(mroz.lwage)
b_ols_man = cov_xy / var_x
println("b_ols_man = $b_ols_man\n")

# IV slope parameter manually:
b_iv_man = cov_yz / cov_xz
println("b_iv_man = $b_iv_man\n")

# OLS automatically:
reg_ols = lm(@formula(lwage ~ educ), mroz)
table_ols = coeftable(reg_ols)
println("table_ols: \n$table_ols\n")

# IV automatically:
reg_iv = fit(EconometricModel,
    @formula(lwage ~ (educ ~ fatheduc)), mroz)
table_iv = coeftable(reg_iv)
println("table_iv: \n$table_iv")
```

Output of Script 15.1: Example-15-1.jl

```
b_ols_man = 0.1086486551746753

b_iv_man = 0.05917347999936603

table_ols:

                Coef.  Std. Error      t  Pr(>|t|)   Lower 95%  Upper 95%

(Intercept)  -0.185197   0.185226   -1.00    0.3180  -0.549267    0.178874
educ          0.108649   0.0143998   7.55    <1e-12   0.0803451   0.136952


table_iv:

                PE          SE      t-value  Pr > |t|        2.50%    97.50%

(Intercept)  0.441103   0.446626   0.987634    0.3239  -0.436768    1.31897
educ         0.0591735  0.0351831  1.68187     0.0933  -0.00998105  0.128328
```

15.2. More Exogenous Regressors

The IV approach can easily be generalized to include additional exogenous regressors, i.e. regressors that are assumed to be unrelated to the error term. In the formula specification of **fit(EconometricModel,...)**, the exogenous regressor(s) **x_exg**, the endogenous regressor(s) **x_end** and instruments **z** are provided in the following way:

```
y ~  x_exg + (x_end ~ z)
```

Wooldridge, Example 15.4: Using College Proximity as an IV for Education

In Script 15.2 (Example-15-4.jl), we use CARD to estimate the return to education. Education is allowed to be endogenous and instrumented with the dummy variable **nearc4** which indicates whether the individual grew up close to a college. In addition, we control for experience, race, and regional information. These variables are assumed to be exogenous and act as their own instruments.
We first check for relevance by regressing the endogenous independent variable **educ** on all exogenous variables including the instrument **nearc4**. Its parameter is highly significantly different from zero, so relevance is supported. We then estimate the log wage equation with OLS and IV.

Script 15.2: Example-15-4.jl

```
using WooldridgeDatasets, GLM, DataFrames, Econometrics

card = DataFrame(wooldridge("card"))

# checking for relevance with reduced form:
reg_redf = lm(@formula(educ ~ nearc4 + exper + (exper^2) + black +
                              smsa + south + smsa66 + reg662 +
                              reg663 + reg664 + reg665 + reg666 +
                              reg667 + reg668 + reg669), card)
table_redf = coeftable(reg_redf)
println("table_redf: \n$table_redf\n")
```

```
# OLS:
reg_ols = lm(@formula(log(wage) ~ educ + exper + (exper^2) + black +
                                  smsa + south + smsa66 + reg662 +
                                  reg663 + reg664 + reg665 + reg666 +
                                  reg667 + reg668 + reg669), card)
table_ols = coeftable(reg_ols)
println("table_ols: \n$table_ols\n")


# IV automatically:
reg_iv = fit(EconometricModel,
    @formula(log(wage) ~ exper + (exper^2) + black + smsa +
                        south + smsa66 + reg662 + reg663 +
                        reg664 + reg665 + reg666 + reg667 +
                        reg668 + reg669 + (educ ~ nearc4)), card)
table_iv = coeftable(reg_iv)
println("table_iv: \n$table_iv")
```

Output of Script 15.2: Example-15-4.jl

```
table_redf:

                   Coef.  Std. Error       t  Pr(>|t|)    Lower 95%    Upper 95%

(Intercept)  16.6383       0.24063      69.14    <1e-99  16.1664      17.1101
nearc4        0.319899     0.0878638     3.64    0.0003   0.147619     0.492179
exper        -0.412533     0.0336996   -12.24    <1e-32  -0.47861     -0.346457
exper ^ 2     0.000868574  0.00165038    0.53    0.5987  -0.00236742   0.00410457
black        -0.935529     0.0937348    -9.98    <1e-22  -1.11932     -0.751738
smsa          0.402182     0.104811      3.84    0.0001   0.196673     0.607692
south        -0.0516126    0.135428     -0.38    0.7032  -0.317155     0.21393
smsa66        0.0254805    0.105769      0.24    0.8096  -0.181907     0.232868
reg662       -0.0786363    0.187115     -0.42    0.6743  -0.445524     0.288251
reg663       -0.027939     0.183375     -0.15    0.8789  -0.387492     0.331614
reg664        0.117182     0.217253      0.54    0.5897  -0.308798     0.543162
reg665       -0.272616     0.21842      -1.25    0.2121  -0.700886     0.155653
reg666       -0.302815     0.237071     -1.28    0.2016  -0.767654     0.162024
reg667       -0.216818     0.234388     -0.93    0.3550  -0.676395     0.24276
reg668        0.523891     0.267475      1.96    0.0502  -0.00056176   1.04834
reg669        0.210271     0.202457      1.04    0.2991  -0.186698     0.60724


table_ols:

                  Coef.   Std. Error       t  Pr(>|t|)    Lower 95%    Upper 95%

(Intercept)   4.62081     0.0742327     62.25    <1e-99   4.47525      4.76636
educ          0.0746933   0.00349835    21.35    <1e-93   0.0678339    0.0815527
exper         0.084832    0.00662422    12.81    <1e-35   0.0718435    0.0978205
exper ^ 2    -0.00228704  0.000316626   -7.22    <1e-12  -0.00290787  -0.00166621
black        -0.199012    0.0182483    -10.91    <1e-26  -0.234793    -0.163232
smsa          0.136385    0.0201005      6.79    <1e-10   0.0969724    0.175797
south        -0.147955    0.0259799     -5.69    <1e-07  -0.198895    -0.0970148
smsa66        0.0262417   0.0194477      1.35    0.1773  -0.0118905    0.0643739
reg662        0.0963672   0.0358979      2.68    0.0073   0.0259801    0.166754
reg663        0.14454     0.0351244      4.12    <1e-04   0.0756696    0.21341
reg664        0.0550756   0.0416573      1.32    0.1862  -0.0266043    0.136755
reg665        0.128025    0.0418395      3.06    0.0022   0.0459878    0.210062
reg666        0.140517    0.0452469      3.11    0.0019   0.0517992    0.229236
reg667        0.117981    0.0448025      2.63    0.0085   0.0301343    0.205828
reg668       -0.0564361   0.0512579     -1.10    0.2710  -0.15694      0.0440682
reg669        0.11857     0.0388301      3.05    0.0023   0.0424335    0.194706
```

```
table_iv:

                    PE           SE          t-value  Pr > |t|        2.50%         97.50%

(Intercept)   3.66615     0.924984      3.96348     <1e-04   1.85248        5.47982
exper         0.108271    0.0236625     4.57564     <1e-05   0.0618746      0.154668
exper ^ 2    -0.00233494  0.000333553  -7.0002      <1e-11  -0.00298895    -0.00168092
black        -0.146776    0.0539089    -2.72266     0.0065  -0.252478      -0.0410736
smsa          0.111808    0.0316673     3.53072     0.0004   0.0497165      0.1739
south        -0.144671    0.0272892    -5.30142     <1e-06  -0.198179      -0.091164
smsa66        0.0185311   0.0216122     0.857437    0.3913  -0.0238452      0.0609074
reg662        0.100768    0.037692      2.67345     0.0075   0.0268629      0.174673
reg663        0.148259    0.0368203     4.02655     <1e-04   0.0760632      0.220454
reg664        0.0498971   0.0437471     1.14058     0.2541  -0.0358804      0.135675
reg665        0.146272    0.0470718     3.10742     0.0019   0.0539755      0.238568
reg666        0.162903    0.0519182     3.13768     0.0017   0.0611039      0.264702
reg667        0.134572    0.0494106     2.72355     0.0065   0.0376901      0.231454
reg668       -0.083077    0.0593413    -1.39999     0.1616  -0.199431       0.0332768
reg669        0.107814    0.0418207     2.57801     0.0100   0.0258141      0.189814
educ          0.131504    0.0549729     2.39216     0.0168   0.0237154      0.239292
```

15.3. Two Stage Least Squares

Two stage least squares (2SLS) is a general approach for IV estimation when we have one or more endogenous regressors and at least as many additional instrumental variables. Consider the regression model

$$y_1 = \beta_0 + \beta_1 y_2 + \beta_2 y_3 + \beta_3 z_1 + \beta_4 z_2 + \beta_5 z_3 + u_1. \tag{15.3}$$

The regressors y_2 and y_3 are potentially correlated with the error term u_1, the regressors z_1, z_2, and z_3 are assumed to be exogenous. Because we have two endogenous regressors, we need at least two additional instrumental variables, say z_4 and z_5.

The name of 2SLS comes from the fact that it can be performed in two stages of OLS regressions:

(1) Separately regress y_2 and y_3 on z_1 through z_5. Obtain fitted values $\hat{y}_2$ and $\hat{y}_3$.

(2) Regress y_1 on $\hat{y}_2$, $\hat{y}_3$, and z_1 through z_3.

If the instruments are valid, this will give consistent estimates of the parameters β_0 through β_5. Generalizing this to more endogenous regressors and instrumental variables is obvious.

This procedure can of course easily be implemented using **`lm`** in **`GLM`**, remembering that fitted values are obtained by **`predict`**. One of the problems of this manual approach is that the resulting variance-covariance matrix and analyses based on them are invalid. Conveniently, **`fit(EconometricModel,...)`** will automatically do these calculations and calculate correct standard errors and the like.

Wooldridge, Example 15.5: Return to Education for Married Women

We continue Example 15.1 and still want to estimate the return to education for women using the data in `MROZ`. Now, we use both mother's and father's education as instruments for their own education.
In Script 15.3 (`Example-15-5.jl`), we obtain 2SLS estimates in two ways: First, we do both stages manually, including fitted education as **`educ_fitted`** as a regressor in the second stage. **`Econometrics`** does this automatically and delivers the same parameter estimates as the output table reveals. But the standard errors differ slightly because the manual two stage version did not correct them.

Script 15.3: Example-15-5.jl

```
using WooldridgeDatasets, GLM, DataFrames, Econometrics

mroz_wm = DataFrame(wooldridge("mroz"))

# restrict to non-missing wage observations:
mroz = mroz_wm[.!ismissing.(mroz_wm.wage), :]

# 1st stage (reduced form):
reg_redf = lm(@formula(educ ~ exper + (exper^2) +
                              motheduc + fatheduc), mroz)
mroz.educ_fitted = predict(reg_redf)
table_redf = coeftable(reg_redf)
println("table_redf: \n$table_redf\n")

# 2nd stage:
reg_secstg = lm(@formula(log(wage) ~ educ_fitted + exper +
                                     (exper^2)), mroz)
table_reg_secstg = coeftable(reg_secstg)
println("table_reg_secstg: \n$table_reg_secstg\n")

# IV automatically:
reg_iv = fit(EconometricModel,
    @formula(log(wage) ~ exper + (exper^2) +
                         (educ ~ motheduc + fatheduc)), mroz)
table_iv = coeftable(reg_iv)
println("table_iv: \n$table_iv")
```

Output of Script 15.3: Example-15-5.jl

```
table_redf:

                  Coef.   Std. Error      t  Pr(>|t|)    Lower 95%   Upper 95%

(Intercept)    9.10264     0.426561    21.34    <1e-68    8.2642      9.94108
exper          0.0452254   0.0402507    1.12    0.2618   -0.0338909   0.124342
exper ^ 2     -0.00100909  0.00120334  -0.84    0.4022   -0.00337437  0.00135619
motheduc       0.157597    0.0358941    4.39    <1e-04    0.087044    0.22815
fatheduc       0.189548    0.0337565    5.62    <1e-07    0.123197    0.2559


table_reg_secstg:

                  Coef.   Std. Error      t  Pr(>|t|)    Lower 95%    Upper 95%

(Intercept)    0.0481003   0.419756     0.11    0.9088   -0.776962     0.873163
educ_fitted    0.0613966   0.0329624    1.86    0.0632   -0.00339334   0.126187
exper          0.0441704   0.0140844    3.14    0.0018    0.0164865    0.0718543
exper ^ 2     -0.00089897  0.00042118  -2.13    0.0334   -0.00172683  -7.11091e-5


table_iv:

                  PE           SE        t-value  Pr > |t|        2.50%        97.50%

(Intercept)    0.0481003   0.401276      0.119868    0.9046  -0.740648      0.836848
exper          0.0441704   0.0134643     3.28056     0.0011   0.017705      0.0706358
exper ^ 2     -0.00089897  0.000402636  -2.23271     0.0261  -0.00169039   -0.000107547
educ           0.0613966   0.0315111     1.94841     0.0520  -0.000541636   0.123335
```

15.4. Testing for Exogeneity of the Regressors

There is another way to get the same IV parameter estimates as with 2SLS. In the same setup as above, this "control function approach" also consists of two stages:

(1) Like in 2SLS, regress y_2 and y_3 on z_1 through z_5. Obtain residuals $\hat{v}_2$ and $\hat{v}_3$ instead of fitted values $\hat{y}_2$ and $\hat{y}_3$.

(2) Regress y_1 on y_2, y_3, z_1, z_2, z_3, and the first stage residuals $\hat{v}_2$ and $\hat{v}_3$.

This approach is as simple to implement as 2SLS and will also result in the same parameter estimates and invalid OLS standard errors in the second stage (unless the dubious regressors y_2 and y_3 are in fact exogenous).

After this second stage regression, we can test for exogeneity in a simple way assuming the instruments are valid. We just need to do a t or F test of the null hypothesis that the parameters of the first-stage residuals are equal to zero. If we reject this hypothesis, this indicates endogeneity of y_2 and y_3.

Wooldridge, Example 15.7: Return to Education for Married Women

In Script 15.4 (`Example-15-7.jl`), we continue Example 15.5 using the control function approach. Again, we use both mother's and father's education as instruments. The first stage regression is identical as in Script 15.3 (`Example-15-5.jl`). The second stage adds the first stage residuals to the original list of regressors. The parameter estimates are identical to both the manual 2SLS and the automatic results. We can perform a t test based on the regression table as a test for exogeneity. Here, $t = \frac{0.058}{0.035} \approx 1.67$ with a two-sided p value of $p = 0.095$, indicating a marginally significant evidence for endogeneity.

Script 15.4: `Example-15-7.jl`

```
using WooldridgeDatasets, GLM, DataFrames

mroz_wm = DataFrame(wooldridge("mroz"))

# restrict to non-missing wage observations:
mroz = mroz_wm[.!ismissing.(mroz_wm.wage), :]

# 1st stage (reduced form):
reg_redf = lm(@formula(educ ~ exper + (exper^2) +
                                motheduc + fatheduc), mroz)
mroz.resid = residuals(reg_redf)

# 2nd stage:
reg_secstg = lm(@formula(log(wage) ~ resid + educ +
                                       exper + (exper^2)), mroz)
table_reg_secstg = coeftable(reg_secstg)
println("table_reg_secstg: \n$table_reg_secstg")
```

Output of Script 15.4: `Example-15-7.jl`

```
table_reg_secstg:

                   Coef.    Std. Error      t  Pr(>|t|)     Lower 95%     Upper 95%

(Intercept)   0.0481003   0.394575      0.12    0.9030  -0.727472      0.823673
resid         0.0581666   0.0348073     1.67    0.0954  -0.0102502     0.126583
educ          0.0613966   0.0309849     1.98    0.0482   0.000492999   0.1223
exper         0.0441704   0.0132394     3.34    0.0009   0.0181471     0.0701937
exper ^ 2    -0.00089897  0.000395913  -2.27    0.0237  -0.00167717   -0.000120767
```

15.5. Testing Overidentifying Restrictions

If we have more instruments than endogenous variables, we can use either all or only some of them. If all are valid, using all improves the accuracy of the 2SLS estimator and reduces its standard errors. If the exogeneity of some is dubious, including them might cause inconsistency. It is therefore useful to test for the exogeneity of a set of dubious instruments if we have another (large enough) set that is undoubtedly exogenous. The procedure is described by Wooldridge (2019, Section 15.5):

(1) Estimate the model by 2SLS and obtain residuals $\hat{u}_1$.

(2) Regress $\hat{u}_1$ on all exogenous variables and calculate R_1^2.

(3) The test statistic nR_1^2 is asymptotically distributed as χ_q^2, where q is the number of *overidentifying* restrictions, i.e. number of instruments minus number of endogenous regressors.

Wooldridge, Example 15.8: Return to Education for Married Women

We will again use the data and model of Examples 15.5 and 15.7. Script 15.5 (`Example-15-8.jl`) estimates the model using **`fit(EconometricModel,...)`**. The results are stored in variable **`reg_iv`**. We then run the auxiliary regression and compute its R^2 as **`R2`**. The test statistic **`teststat`** is computed to be 0.378. We also compute the p value from the χ_1^2 distribution. We cannot reject exogeneity of the instruments using this test. But be aware of the fact that the underlying assumption that at least one instrument is valid might be violated here.

Script 15.5: **`Example-15-8.jl`**

```
using WooldridgeDatasets, GLM, DataFrames, Econometrics, Distributions

mroz_wm = DataFrame(wooldridge("mroz"))

# restrict to non-missing wage observations:
mroz = mroz_wm[.!ismissing.(mroz_wm.wage), :]

# IV regression:
reg_iv = fit(EconometricModel,
    @formula(log(wage) ~ exper + (exper^2) +
                         (educ ~ motheduc + fatheduc)), mroz)
table_iv = coeftable(reg_iv)
println("table_iv: \n$table_iv\n")

# auxiliary regression:
mroz.resid_iv = residuals(reg_iv)
reg_aux = lm(@formula(resid_iv ~ exper + (exper^2) +
                                 motheduc + fatheduc), mroz)

# calculations for test:
R2 = r2(reg_aux)
n = nobs(reg_aux)
teststat = n * R2
pval = 1 - cdf(Chisq(1), teststat)

println("R2 = $R2\n")
println("n = $n\n")
println("teststat = $teststat\n")
println("pval = $pval")
```

Output of Script 15.5: `Example-15-8.jl`

```
table_iv:

                  PE            SE         t-value  Pr > |t|         2.50%        97.50%

(Intercept)   0.0481003   0.401276      0.119868    0.9046  -0.740648      0.836848
exper         0.0441704   0.0134643     3.28056     0.0011   0.017705      0.0706358
exper ^ 2    -0.00089897  0.000402636  -2.23271     0.0261  -0.00169039   -0.000107547
educ          0.0613966   0.0315111     1.94841     0.0520  -0.000541636   0.123335


R2 = 0.0008833444088020004

n = 428.0

teststat = 0.37807140696725616

pval = 0.5386371981604867
```

15.6. Instrumental Variables with Panel Data

Instrumental variables can be used for panel data, too. In this way, we can get rid of time-constant individual heterogeneity by first differencing or within transformations and then fix remaining endogeneity problems with instrumental variables.

We know how to get panel data estimates using OLS on the transformed data, so we can easily use IV as before. Script 15.6 (`Example-15-10.jl`) demonstrates such a procedure. Also note that the **`Econometrics`** package supports IV estimation with other implemented panel data methods. For example, the following works as expected:

```
fit(RandomEffectsEstimator, @formula(y ~  x_exg + (x_end ~ z)), data,
             panel = :id,
             time = :t)
```

Wooldridge, Example 15.10: Job Training and Worker Productivity

We use the data set `JTRAIN` to estimate the effect of job training **`hrsemp`** on the scrap rate. In Script 15.6 (`Example-15-10.jl`), we load the data, choose a subset of the years 1987 and 1988 with **`subset`** and store the data with correct index variables **`fcode`** and **`year`**, see Section 14.1. Then we estimate the parameters using first-differencing with the instrumental variable **`grant`**.

Script 15.6: `Example-15-10.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Econometrics

jtrain = DataFrame(wooldridge("jtrain"))

# define panel data (for 1987 and 1988 only) and sort:
jtrain_8788 = subset(jtrain, :year => ByRow(<=(1988)))
sort!(jtrain_8788, [:fcode, :year])

# manual computation of deviations of entity means:
grouped_df = groupby(jtrain_8788, :fcode)
diff_df = DataFrame(fcode=unique(jtrain_8788.fcode))
diff_df.lscrap_diff1 = combine(grouped_df, :lscrap => diff).lscrap_diff
diff_df.hrsemp_diff1 = combine(grouped_df, :hrsemp => diff).hrsemp_diff
diff_df.grant_diff1 = combine(grouped_df, :grant => diff).grant_diff

# IV regression:
reg_iv = fit(EconometricModel,
    @formula(lscrap_diff1 ~ (hrsemp_diff1 ~ grant_diff1)), diff_df)
table_iv = coeftable(reg_iv)
println("table_iv: \n$table_iv")
```

Output of Script 15.6: `Example-15-10.jl`

```
table_iv:

                      PE          SE       t-value  Pr > |t|       2.50%      97.50%

(Intercept)   -0.0326684  0.128454    -0.254321    0.8005  -0.291898   0.226561
hrsemp_diff1  -0.0141532  0.00800841  -1.76729     0.0844  -0.0303148  0.00200846
```

16. Simultaneous Equations Models

In simultaneous equations models (SEM), both the dependent variable and at least one regressor are determined jointly. This leads to an endogeneity problem and inconsistent OLS parameter estimators. The main challenge for successfully using SEM is to specify a sensible model and make sure it is identified, see Wooldridge (2019, Sections 16.1–16.3). We briefly introduce a general model and the notation in Section 16.1.

As discussed in Chapter 15, 2SLS regression can solve endogeneity problems if there are enough exogenous instrumental variables. This also works in the setting of SEM, an example is given in Section 16.2. We implement more advanced estimation commands using the module **`linearmodels`** in *Python*, because so far there is no implementation in *Julia*.[1] We will show this for three-stage-least-squares (3SLS) estimation in Section 16.3 with the help of the **`PyCall`** package. Check Section 1.2.5 to review the installation of *Python* modules for the use in **`PyCall`**.

16.1. Setup and Notation

Consider the general SEM with q endogenous variables $y_1, \dots, y_q$ and k exogenous variables $x_1, \dots, x_k$. The system of equations is:

$$
\begin{aligned}
y_1 &= \alpha_{12} y_2 + \alpha_{13} y_3 + \cdots + \alpha_{1q} y_q \qquad + \beta_{10} + \beta_{11} x_1 + \cdots + \beta_{1k} x_k + u_1 \\
y_2 &= \alpha_{21} y_1 + \alpha_{23} y_3 + \cdots + \alpha_{2q} y_q \qquad + \beta_{20} + \beta_{21} x_1 + \cdots + \beta_{2k} x_k + u_2 \\
&\vdots \\
y_q &= \alpha_{q1} y_1 + \alpha_{q2} y_2 + \cdots + \alpha_{qq-1} y_{q-1} + \beta_{q0} + \beta_{q1} x_1 + \cdots + \beta_{qk} x_k + u_q
\end{aligned}
$$

As discussed in more detail in Wooldridge (2019, Section 16), this system is not identified without restrictions on the parameters. The order condition for identification of any equation is that if we have m included endogenous regressors (i.e. α parameters that are not restricted to 0), we need to exclude at least m exogenous regressors (i.e. restrict their β parameters to 0). They can then be used as instrumental variables.

[1] For details, see the module documentation `https://bashtage.github.io/linearmodels/` or Chapter 16 in Heiss and Brunner (2020).

Wooldridge, Example 16.3: Labor Supply of Married, Working Women

We have the two endogenous variables `hours` and `wage` which influence each other.

$$\begin{aligned}\texttt{hours} &= \alpha_{12}\log(\texttt{wage}) + \beta_{10} + \beta_{11}\texttt{educ} + \beta_{12}\texttt{age} + \beta_{13}\texttt{kidslt6} + \beta_{14}\texttt{nwifeinc} \\ &\qquad + \beta_{15}\texttt{exper} + \beta_{16}\texttt{exper}^2 + u_1 \\ \log(\texttt{wage}) &= \alpha_{21}\texttt{hours} \qquad + \beta_{20} + \beta_{21}\texttt{educ} + \beta_{22}\texttt{age} + \beta_{23}\texttt{kidslt6} + \beta_{24}\texttt{nwifeinc} \\ &\qquad + \beta_{25}\texttt{exper} + \beta_{26}\texttt{exper}^2 + u_2\end{aligned}$$

For both equations to be identified, we have to exclude at least one exogenous regressor from each equation. Wooldridge (2019) discusses a model in which we restrict $\beta_{15} = \beta_{16} = 0$ in the first and $\beta_{22} = \beta_{23} = \beta_{24} = 0$ in the second equation.

16.2. Estimation by 2SLS

Estimation of each equation separately by 2SLS is straightforward once we have set up the system and ensured identification. The excluded regressors in each equation serve as instrumental variables. As shown in Chapter 15, the command **`fit(EconometricModel,...)`** from the package **`Econometrics`** provides convenient 2SLS estimation. Script 16.1 (`Example-16-5-2SLS-1.jl`) gives an example and Script 16.2 (`Example-16-5-2SLS-2.jl`) shows the implementation with *Python*'s **`linearmodels`**. The naming in the output is a bit hard to read, which is due to the cumbersome passing of matrices.[2] A pure *Python* solution does not have this problem.

Wooldridge, Example 16.5: Labor Supply of Married, Working Women

Script 16.1 (`Example-16-5-2SLS-1.jl`) estimates the parameters of the two equations from Example 16.3 separately using **`Econometrics`**. Script 16.2 (`Example-16-5-2SLS-2.jl`) repeats the exercise using the method **`IV2SLS`** and gives identical point estimates.

[2]The module **`linearmodels`** also supports formula syntax. However passing the formula via **`PyCall`** does not work at this time.

Script 16.1: Example-16-5-2SLS-1.jl

```
using WooldridgeDatasets, GLM, DataFrames, Econometrics, Statistics

mroz_wm = DataFrame(wooldridge("mroz"))

# restrict to non-missing wage observations:
mroz = mroz_wm[.!ismissing.(mroz_wm.wage), :]

# 2SLS regressions:
reg_iv1 = fit(EconometricModel,
    @formula(hours ~ educ + age + kidslt6 + nwifeinc +
                     (log(wage) ~ exper + (exper^2))), mroz)
table_iv1 = coeftable(reg_iv1)
println("table_iv1: \n$table_iv1\n")

reg_iv2 = fit(EconometricModel,
    @formula(log(wage) ~ educ + exper + (exper^2) +
                         (hours ~ age + kidslt6 + nwifeinc)), mroz)
table_iv2 = coeftable(reg_iv2)
println("table_iv2: \n$table_iv2\n")

cor_u1u2 = cor(residuals(reg_iv1), residuals(reg_iv2))
println("cor_u1u2 =$cor_u1u2")
```

Output of Script 16.1: Example-16-5-2SLS-1.jl

```
table_iv1:

                  PE         SE        t-value  Pr > |t|      2.50%      97.50%

(Intercept)  2225.66     575.931     3.86446     0.0001  1093.6     3357.73
educ         -183.751     59.2404   -3.10179     0.0021  -300.196    -67.3068
age            -7.80609    9.40032  -0.830407    0.4068   -26.2836    10.6714
kidslt6      -198.154    183.364    -1.08066     0.2805  -558.58     162.272
nwifeinc      -10.1696     6.63047  -1.53377     0.1258   -23.2026     2.86346
log(wage)    1639.56     471.695     3.47588     0.0006   712.379   2566.73


table_iv2:

                  PE            SE         t-value  Pr > |t|         2.50%       97.50%

(Intercept)  -0.655725     0.338993     -1.93434     0.0537  -1.32206       0.0106079
educ          0.11033      0.0155797     7.08165     <1e-11   0.0797061     0.140954
exper         0.0345824    0.019561      1.76792     0.0778  -0.00386739    0.0730321
exper ^ 2    -0.000705769  0.000455699  -1.54876     0.1222  -0.0016015     0.000189966
hours         0.0001259    0.000255518   0.492725    0.6225  -0.000376354   0.000628154


cor_u1u2 =-0.9037694196299592
```

Script 16.2: `Example-16-5-2SLS-2.jl`

```
using WooldridgeDatasets, GLM, DataFrames, PyCall
include("../03/getMats.jl")

# install Python's linearmodels with: using Conda; Conda.add("linearmodels")
iv = pyimport("linearmodels.iv")

mroz_wm = DataFrame(wooldridge("mroz"))

# restrict to non-missing wage observations:
mroz = mroz_wm[.!ismissing.(mroz_wm.wage), :]

# prepare for equation 1:
f1 = @formula(hours ~ 1 + educ + age + kidslt6 + nwifeinc)
yexog = getMats(f1, mroz)
y_eq1 = yexog[1]
exog_mat_eq1 = yexog[2]

f2 = @formula(1 ~ 0 + log(wage))
endo_mat_eq1 = getMats(f2, mroz)[2]

f3 = @formula(1 ~ 0 + exper + (exper^2))
iv_mat_eq1 = getMats(f3, mroz)[2]

# prepare for equation 2:
f1 = @formula(log(wage) ~ 1 + educ + exper + (exper^2))
yexog = getMats(f1, mroz)
y_eq2 = yexog[1]
exog_mat_eq2 = yexog[2]

f2 = @formula(1 ~ 0 + hours)
endo_mat_eq2 = getMats(f2, mroz)[2]

f3 = @formula(1 ~ 0 + age + kidslt6 + nwifeinc)
iv_mat_eq2 = getMats(f3, mroz)[2]

# use Python's linearmodels:
reg_iv1 = iv.IV2SLS(y_eq1, exog_mat_eq1, endo_mat_eq1, iv_mat_eq1)
results_iv1 = reg_iv1.fit(cov_type="unadjusted", debiased=true)
println("results_iv1: \n$results_iv1\n")

reg_iv2 = iv.IV2SLS(y_eq2, exog_mat_eq2, endo_mat_eq2, iv_mat_eq2)
results_iv2 = reg_iv2.fit(cov_type="unadjusted", debiased=true)
println("results_iv2: \n$results_iv2")
```

Output of Script 16.2: `Example-16-5-2SLS-2.jl`

```
results_iv1:
PyObject                          IV-2SLS Estimation Summary
==============================================================================
Dep. Variable:              dependent   R-squared:                     -2.0076
Estimator:                    IV-2SLS   Adj. R-squared:                -2.0433
No. Observations:                 428   F-statistic:                    3.4410
Date:                Tue, Mar 21 2023   P-value (F-stat)                0.0046
Time:                        08:39:59   Distribution:                 F(5,422)
Cov. Estimator:            unadjusted

                             Parameter Estimates
==============================================================================
            Parameter  Std. Err.     T-stat    P-value    Lower CI    Upper CI
------------------------------------------------------------------------------
exog.0         2225.7     574.56     3.8737     0.0001      1096.3      3355.0
exog.1        -183.75     59.100    -3.1092     0.0020     -299.92     -67.585
exog.2        -7.8061     9.3780    -0.8324     0.4057     -26.240      10.627
exog.3        -198.15     182.93    -1.0832     0.2793     -557.72      161.41
exog.4        -10.170     6.6147    -1.5374     0.1249     -23.172      2.8324
endog          1639.6     470.58     3.4841     0.0005      714.59      2564.5
==============================================================================

Endogenous: endog
Instruments: instruments.0, instruments.1
Unadjusted Covariance (Homoskedastic)
Debiased: True
IVResults, id: 0x2c2f0f430

results_iv2:
PyObject                          IV-2SLS Estimation Summary
==============================================================================
Dep. Variable:              dependent   R-squared:                      0.1257
Estimator:                    IV-2SLS   Adj. R-squared:                 0.1174
No. Observations:                 428   F-statistic:                    19.028
Date:                Tue, Mar 21 2023   P-value (F-stat)                0.0000
Time:                        08:39:59   Distribution:                 F(4,423)
Cov. Estimator:            unadjusted

                             Parameter Estimates
==============================================================================
            Parameter  Std. Err.     T-stat    P-value    Lower CI    Upper CI
------------------------------------------------------------------------------
exog.0        -0.6557     0.3378    -1.9412     0.0529     -1.3197      0.0082
exog.1         0.1103     0.0155     7.1069     0.0000      0.0798      0.1408
exog.2         0.0346     0.0195     1.7742     0.0767     -0.0037      0.0729
exog.3        -0.0007     0.0005    -1.5543     0.1209     -0.0016      0.0002
endog          0.0001     0.0003     0.4945     0.6212     -0.0004      0.0006
==============================================================================

Endogenous: endog
Instruments: instruments.0, instruments.1, instruments.2
Unadjusted Covariance (Homoskedastic)
Debiased: True
IVResults, id: 0x2c2f4e800
```

16.3. Outlook: Estimation by 3SLS

An interesting piece of information in Script 16.1 (Example-16-5-2SLS-1.jl) is the correlation between the residuals of the equations. In the example, it is reported to be a substantially negative -0.90. We can account for the correlation between the error terms to derive a potentially more efficient parameter estimator than 2SLS. Without going into details here, the three stage least squares (3SLS) estimator adds another stage to 2SLS by estimating the correlation and accounting for it using a FGLS approach. For a detailed discussion of this and related methods, see for example Wooldridge (2010, Chapter 8).

Using 3SLS in *Python*'s **linearmodels** is simple: The function **IV3SLS** is all we need as the output of Script 16.3 (Example-16-5-3SLS.jl) shows. Again, **PyCall** helps us to call this function within *Julia*.

Script 16.3: Example-16-5-3SLS.jl

```
using WooldridgeDatasets, GLM, DataFrames, PyCall
include("../03/getMats.jl")
iv3 = pyimport("linearmodels.system")

mroz_wm = DataFrame(wooldridge("mroz"))

# restrict to non-missing wage observations:
mroz = mroz_wm[.!ismissing.(mroz_wm.wage), :]

# prepare for equation 1:
f1 = @formula(hours ~ 1 + educ + age + kidslt6 + nwifeinc)
yexog = getMats(f1, mroz)
y_eq1 = yexog[1]
exog_mat_eq1 = yexog[2]

f2 = @formula(1 ~ 0 + log(wage))
endo_mat_eq1 = getMats(f2, mroz)[2]

f3 = @formula(1 ~ 0 + exper + (exper^2))
iv_mat_eq1 = getMats(f3, mroz)[2]

# prepare for equation 2:
f1 = @formula(log(wage) ~ 1 + educ + exper + (exper^2))
yexog = getMats(f1, mroz)
y_eq2 = yexog[1]
exog_mat_eq2 = yexog[2]

f2 = @formula(1 ~ 0 + hours)
endo_mat_eq2 = getMats(f2, mroz)[2]

f3 = @formula(1 ~ 0 + age + kidslt6 + nwifeinc)
iv_mat_eq2 = getMats(f3, mroz)[2]

# use Python's linearmodels:
reg_3sls = iv3.IV3SLS(Dict([
    ("eq1", (y_eq1, exog_mat_eq1, endo_mat_eq1, iv_mat_eq1)),
    ("eq2", (y_eq2, exog_mat_eq2, endo_mat_eq2, iv_mat_eq2))]))
results_3sls = reg_3sls.fit(cov_type="unadjusted", debiased=true)
println("results_3sls: \n$results_3sls")
```

Output of Script 16.3: `Example-16-5-3SLS.jl`

```
results_3sls:
PyObject                           System GLS Estimation Summary
===================================================================================
Estimator:                          GLS   Overall R-squared:                -2.3957
No. Equations.:                       2   McElroy's R-squared:               0.7846
No. Observations:                   428   Judge's (OLS) R-squared:          -2.3957
Date:                  Tue, Mar 21 2023   Berndt's R-squared:                0.5181
Time:                          08:39:59   Dhrymes's R-squared:              -2.3957
                                          Cov. Estimator:                unadjusted
                                          Num. Constraints:                    None
                Equation: eq1, Dependent Variable: dependent_0
==============================================================================
            Parameter  Std. Err.     T-stat    P-value    Lower CI    Upper CI
------------------------------------------------------------------------------
exog_0.0       2305.9     511.54     4.5077     0.0000      1300.4      3311.3
exog_0.1      -212.82     53.727    -3.9611     0.0001     -318.43     -107.21
exog_0.2      -9.5150     7.9609    -1.1952     0.2327     -25.163      6.1331
exog_0.3      -192.36     150.92    -1.2746     0.2032     -489.00      104.28
exog_0.4      -0.1770     3.5836    -0.0494     0.9606     -7.2210      6.8670
endog_0        1781.9     439.88     4.0509     0.0001      917.30      2646.6

====================
    Instruments
--------------------
instr_0.0, instr_0.1

                Equation: eq2, Dependent Variable: dependent_1
==============================================================================
            Parameter  Std. Err.     T-stat    P-value    Lower CI    Upper CI
------------------------------------------------------------------------------
exog_1.0      -0.6939     0.3360    -2.0653     0.0395     -1.3543     -0.0335
exog_1.1       0.1127     0.0154     7.3355     0.0000      0.0825      0.1429
exog_1.2       0.0214     0.0154     1.3929     0.1644     -0.0088      0.0517
exog_1.3      -0.0003     0.0003    -1.1303     0.2590     -0.0008      0.0002
endog_1        0.0002     0.0002     0.7707     0.4413     -0.0003      0.0007
===============================
          Instruments
-------------------------------
instr_1.0, instr_1.1, instr_1.2
-------------------------------

Covariance Estimator:
Homoskedastic (Unadjusted) Covariance (Debiased: True, GLS: True)
SystemResults, id: 0x2c2f74c70
```

17. Limited Dependent Variable Models and Sample Selection Corrections

A limited dependent variable (LDV) can only take a limited set of values. An extreme case are binary variables that can only take two values. We already used such dummy variables as regressors in Chapter 7. Section 17.1 discusses how to use them as dependent variables. Another example for LDV are counts that take only non-negative integers, they are covered in Section 17.2. Similarly, Tobit models discussed in Section 17.3 deal with dependent variables that can only take positive values (or are restricted in a similar way), but are otherwise continuous.

The Sections 17.4 and 17.5 are concerned with continuous dependent variables but are not perfectly observed. For some units of the censored, truncated, or selected observations we only know that they are above or below a certain threshold or we don't know anything about them.

17.1. Binary Responses

Binary dependent variables are frequently studied in applied econometrics. Because a dummy variable y can only take the values 0 and 1, its (conditional) expected value is equal to the (conditional) probability that $y = 1$:

$$\begin{aligned} \mathrm{E}(y|\mathbf{x}) &= 0 \cdot \mathrm{P}(y=0|\mathbf{x}) + 1 \cdot \mathrm{P}(y=1|\mathbf{x}) \\ &= \mathrm{P}(y=1|\mathbf{x}) \end{aligned} \tag{17.1}$$

So when we study the conditional mean, it makes sense to think about it as the probability of outcome $y = 1$. Likewise, the predicted value $\hat{y}$ should be thought of as a predicted probability.

17.1.1. Linear Probability Models

If a dummy variable is used as the dependent variable y, we can still use OLS to estimate its relation to the regressors $\mathbf{x}$. These linear probability models are covered by Wooldridge (2019) in Section 7.5. If we write the usual linear regression model

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k \tag{17.2}$$

and make the usual assumptions, especially MLR.4: $E(u|\mathbf{x}) = 0$, this implies for the conditional mean (which is the probability that $y = 1$) and the predicted probabilities:

$$\mathrm{P}(y=1|\mathbf{x}) = \mathrm{E}(y|\mathbf{x}) = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k \tag{17.3}$$

$$\hat{\mathrm{P}}(y=1|\mathbf{x}) = \quad \hat{y} \quad = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_k x_k \tag{17.4}$$

The interpretation of the parameters is straightforward: β_j is a measure of the average change in probability of a "success" ($y = 1$) as x_j increases by one unit and the other determinants remain constant. Linear probability models automatically suffer from heteroscedasticity, so with OLS, we should use heteroscedasticity-robust inference, see Section 8.1.

Wooldridge, Example 17.1: Married Women's Labor Force Participation

We study the probability that a woman is in the labor force depending on socio-demographic characteristics. Script 17.1 (`Example-17-1-1.jl`) estimates a linear probability model using the data set `mroz`. The estimated coefficient of **educ** can be interpreted as: an additional year of schooling increases the probability that a woman is in the labor force *ceteris paribus* by 0.038 on average. We used White's robust standard errors as discussed in Chapter 8.

Script 17.1: `Example-17-1-1.jl`

```
using WooldridgeDatasets, GLM, DataFrames
include("../08/calc-white-se.jl")

mroz = DataFrame(wooldridge("mroz"))

# estimate linear probability model:
reg_lin = lm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) +
                             age + kidslt6 + kidsge6), mroz)
hc0 = calc_white_se(reg_lin, mroz)

table_reg_lin = DataFrame(
    coefficients=coeftable(reg_lin).rownms,
    b=round.(coef(reg_lin), digits=5),
    se_white=hc0)
println("table_reg_lin: \n$table_reg_lin")
```

Output of Script 17.1: `Example-17-1-1.jl`

```
table_reg_lin:
8×3 DataFrame
 Row | coefficients  b         se_white
     | String        Float64   Float64
     -----------------------------------------
   1 | (Intercept)    0.58552  0.151449
   2 | nwifeinc      -0.00341  0.00151681
   3 | educ           0.038    0.00722734
   4 | exper          0.03949  0.00577907
   5 | exper ^ 2     -0.0006   0.000188992
   6 | age           -0.01609  0.00238623
   7 | kidslt6       -0.26181  0.0316139
   8 | kidsge6        0.01301  0.0134609
```

One problem with linear probability models is that $\mathrm{P}(y = 1|\mathbf{x})$ is specified as a linear function of the regressors. By construction, there are (more or less realistic) combinations of regressor values that yield $\hat{y} < 0$ or $\hat{y} > 1$. Since these are probabilities, this does not really make sense.

As an example, Script 17.2 (`Example-17-1-2.jl`) calculates the predicted values for two women (see Section 6.2 for how to **`predict`** after OLS estimation): Woman 1 is 20 years old, has no work experience, 5 years of education, two children below age 6 and has additional family income of 100,000 USD. Woman 2 is 52 years old, has 30 years of work experience, 17 years of education, no children and no other source of income. The predicted "probability" for woman 1 is -41%, the probability for woman 2 is 104% as can also be easily checked with a calculator.

Script 17.2: `Example-17-1-2.jl`

```
using WooldridgeDatasets, GLM, DataFrames

mroz = DataFrame(wooldridge("mroz"))

# estimate linear probability model:
reg_lin = lm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) +
                              age + kidslt6 + kidsge6), mroz)

# predictions for two "extreme" women:
X_new = DataFrame(nwifeinc=[100, 0], educ=[5, 17],
     exper=[0, 30], age=[20, 52],
     kidslt6=[2, 0], kidsge6=[0, 0])
predictions = round.(predict(reg_lin, X_new), digits=5)

print("predictions = $predictions")
```

Output of Script 17.2: `Example-17-1-2.jl`

```
predictions = [-0.41046, 1.04281]
```

17.1.2. Logit and Probit Models: Estimation

Specialized models for binary responses make sure that the implied probabilities are restricted between 0 and 1. An important class of models specifies the success probability as

$$\mathrm{P}(y=1|\mathbf{x}) = G(\beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k) = G(\mathbf{x}\boldsymbol{\beta}) \tag{17.5}$$

where the "link function" $G(z)$ always returns values between 0 and 1. In the statistics literature, this type of models is often called generalized linear model (GLM) because a linear part $\mathbf{x}\boldsymbol{\beta}$ shows up within the nonlinear function G.

For binary response models, by far the most widely used specifications for G are

- the **probit** model with $G(z) = \Phi(z)$, the standard normal CDF and
- the **logit** model with $G(z) = \Lambda(z) = \frac{\exp(z)}{1+\exp(z)}$, the CDF of the logistic distribution.

Wooldridge (2019, Section 17.1) provides useful discussions of the derivation and interpretation of these models. Here, we are concerned with the practical implementation. In the **`GLM`** package, many generalized linear models can be estimated with the function **`glm`** working similar as **`lm`**. In the following, we will use two of them frequently:

- **`LogitLink`** for the logit model and
- **`ProbitLink`** for the probit model.

Given the data set **`sample`** contains variables **`y`**, **`x1`**, **`x2`**, **`x3`**, with the respective data of our sample, we can estimate these models with the following code:

```
reg_logit = glm(@formula(y ~ x1 + x2 + x3), sample, Binomial(), LogitLink())
reg_probit = glm(@formula(y ~ x1 + x2 + x3), sample, Binomial(), ProbitLink())
```

Maximum likelihood estimation (MLE) of the parameters is done automatically and the **`coeftable`** command gives the regression table. Scripts 17.3 (`Example-17-1-3.jl`) and 17.4 (`Example-17-1-4.jl`) implement the logit and probit model, respectively. The log likelihood value $\mathscr{L}(\hat{\boldsymbol{\beta}})$ can be computed by **`deviance(reg) / -2`** with **`reg`** as the output of the **`glm`** command. We can also calculate $\mathscr{L}_0$, which is the log likelihood of a model with an intercept only.

Scripts 17.3 (`Example-17-1-3.jl`) and 17.4 (`Example-17-1-4.jl`) demonstrate these computations to calculate McFadden's pseudo R-squared as

$$\text{pseudo } R^2 = 1 - \frac{\mathscr{L}(\hat{\beta})}{\mathscr{L}_0}. \tag{17.6}$$

Script 17.3: `Example-17-1-3.jl`

```
using WooldridgeDatasets, GLM, DataFrames

mroz = DataFrame(wooldridge("mroz"))

# estimate logit model:
reg_logit = glm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) + age +
                                kidslt6 + kidsge6),
    mroz, Binomial(), LogitLink())
table_reg_logit = coeftable(reg_logit)
println("table_reg_logit: \n$table_reg_logit\n")

# log likelihood value:
ll = deviance(reg_logit) / -2
println("ll = $ll\n")

# McFadden's pseudo R2:
reg_logit_null = glm(@formula(inlf ~ 1), mroz, Binomial(), LogitLink())
ll_null = deviance(reg_logit_null) / -2
pr2 = 1 - ll / ll_null
println("pr2 = $pr2")
```

Output of Script 17.3: `Example-17-1-3.jl`

```
table_reg_logit:

                  Coef.  Std. Error      z  Pr(>|z|)    Lower 95%    Upper 95%

(Intercept)   0.425452   0.860365     0.49    0.6210  -1.26083      2.11174
nwifeinc     -0.0213452  0.00842138  -2.53    0.0113  -0.0378508   -0.00483957
educ          0.22117    0.0434393    5.09    <1e-06   0.136031     0.30631
exper         0.20587    0.0320567    6.42    <1e-09   0.14304      0.2687
exper ^ 2    -0.0031541  0.00101611  -3.10    0.0019  -0.00514564  -0.00116257
age          -0.0880244  0.0145729   -6.04    <1e-08  -0.116587    -0.059462
kidslt6      -1.44335    0.203583    -7.09    <1e-11  -1.84237     -1.04434
kidsge6       0.0601122  0.0747893    0.80    0.4215  -0.0864721    0.206697


ll = -401.7651511343817

pr2 = 0.2196813748112092
```

Script 17.4: Example-17-1-4.jl

```
using WooldridgeDatasets, GLM, DataFrames

mroz = DataFrame(wooldridge("mroz"))

# estimate probit model:
reg_probit = glm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) + age +
                                 kidslt6 + kidsge6),
    mroz, Binomial(), ProbitLink())
table_reg_probit = coeftable(reg_probit)
println("table_reg_probit: \n$table_reg_probit\n")

# log likelihood value:
ll = deviance(reg_probit) / -2
println("ll=$ll\n")

# McFadden's pseudo R2:
reg_probit_null = glm(@formula(inlf ~ 1), mroz, Binomial(), ProbitLink())
ll_null = deviance(reg_probit_null) / -2
pr2 = 1 - ll / ll_null
println("pr2 = $pr2")
```

Output of Script 17.4: Example-17-1-4.jl

```
table_reg_probit:

                   Coef.    Std. Error      z  Pr(>|z|)   Lower 95%     Upper 95%

(Intercept)   0.270074    0.508078       0.53    0.5950  -0.725741    1.26589
nwifeinc     -0.0120236   0.00493917   -2.43    0.0149  -0.0217042  -0.00234304
educ          0.130904    0.0253987      5.15    <1e-06   0.0811234   0.180685
exper         0.123347    0.0187587      6.58    <1e-10   0.0865808   0.160114
exper ^ 2    -0.00188707  0.000599927  -3.15    0.0017  -0.0030629  -0.000711232
age          -0.0528524   0.00846236   -6.25    <1e-09  -0.0694384  -0.0362665
kidslt6      -0.868325    0.118377     -7.34    <1e-12  -1.10034    -0.636309
kidsge6       0.0360056   0.0440303      0.82    0.4135  -0.0502921   0.122303


ll=-401.3021931756048

pr2 = 0.22058054368360436
```

17.1.3. Inference

The output of the logit or probit results contains a standard regression table with parameters and (asymptotic) standard errors. The next column is labeled **z** instead of **t** in the output of **coeftable**. The interpretation is the same. The difference is that the standard errors only have an asymptotic foundation and the distribution used for calculating p values is the standard normal distribution (which is equal to the t distribution with very large degrees of freedom). The bottom line is that tests for single parameters can be done as before, see Section 4.1.

For testing multiple hypotheses similar to the F test (see Section 4.3), the likelihood ratio test is popular. It is based on comparing the log likelihood values of the unrestricted and the restricted model. The test statistic is

$$LR = 2(\mathscr{L}_{ur} - \mathscr{L}_r) \tag{17.7}$$

where $\mathscr{L}_{ur}$ and $\mathscr{L}_r$ are the log likelihood values of the unrestricted and restricted model, respectively. Under H_0, the LR test statistic is asymptotically distributed as χ^2 with the degrees of freedom equal to the number of restrictions to be tested. The test of overall significance is a special case just like with F tests. The null hypothesis is that all parameters except the constant are equal to zero. With the notation above, the test statistic is

$$LR = 2\big(\mathscr{L}(\hat{\boldsymbol{\beta}}) - \mathscr{L}_0\big). \tag{17.8}$$

For other hypotheses, you can compute LR based on the log likelihood of a restricted model. Script 17.5 (`Example-17-1-5.jl`) implements the test of overall significance for the probit model. It also tests the joint null hypothesis that experience and age are irrelevant.

Script 17.5: `Example-17-1-5.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Distributions

mroz = DataFrame(wooldridge("mroz"))

# estimate probit model:
reg_probit = glm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) + age +
                                 kidslt6 + kidsge6),
    mroz, Binomial(), ProbitLink())
ll = deviance(reg_probit) / -2

# test of overall significance (test statistic and p value):
reg_probit_null = glm(@formula(inlf ~ 1), mroz, Binomial(), ProbitLink())
ll_null = deviance(reg_probit_null) / -2
lr1 = 2 * (ll - ll_null)
pval_all = 1 - cdf(Chisq(7), lr1)
println("lr1 = $lr1\n")
println("pval_all = $pval_all\n")

# likelihood ratio statistic test of H0 (experience and age are irrelevant):
reg_probit_hyp = glm(@formula(inlf ~ nwifeinc + educ + kidslt6 + kidsge6),
    mroz, Binomial(), ProbitLink())
ll_hyp = deviance(reg_probit_hyp) / -2
lr2 = 2 * (ll - ll_hyp)
pval_hyp = 1 - cdf(Chisq(3), lr2)
println("lr2 = $lr2\n")
println("pval_hyp = $pval_hyp")
```

Output of Script 17.5: **Example-17-1-5.jl**

```
lr1 = 227.1420227830812

pval_all = 0.0

lr2 = 127.03401046039562

pval_hyp = 0.0
```

17.1.4. Predictions

The command **predict** can calculate predicted values for the estimation sample ("fitted values") or arbitrary sets of regressor values also for binary response models. Given the results of the **glm** function are stored in the variable **reg**, we can calculate:

- $\hat{y} = G(\mathbf{x}_i\hat{\boldsymbol{\beta}})$ for the estimation sample with **predict(reg)**
- $\hat{y} = G(\mathbf{x}_i\hat{\boldsymbol{\beta}})$ for the regressor values stored in **xpred** with **predict(reg, xpred)**

The predictions for the two hypothetical women introduced in Section 17.1.1 are repeated for the linear probability, logit, and probit models in Script 17.6 (`Example-17-1-6.jl`). Unlike the linear probability model, the predicted probabilities from the logit and probit models remain between 0 and 1.

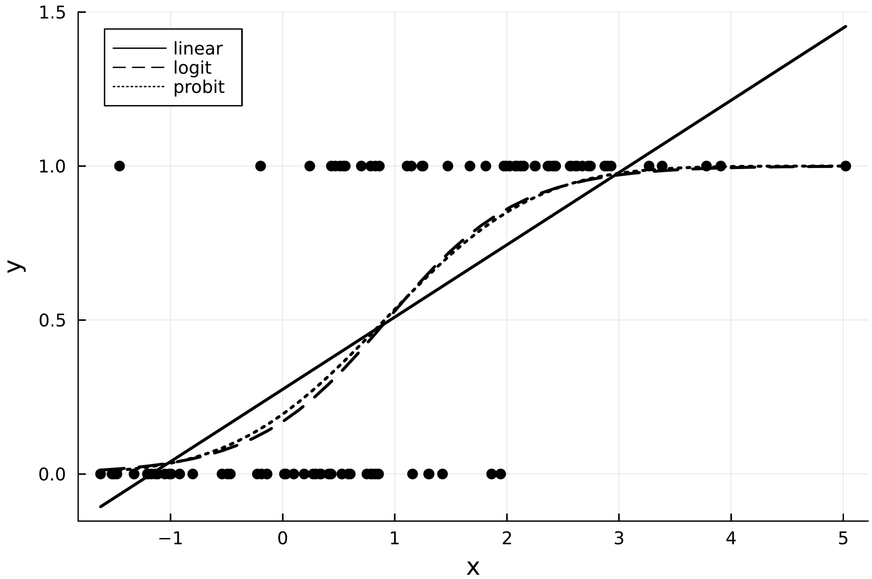
Script 17.6: **Example-17-1-6.jl**

```
using WooldridgeDatasets, GLM, DataFrames

mroz = DataFrame(wooldridge("mroz"))

# estimate models:
reg_lin = lm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) + age +
                             kidslt6 + kidsge6), mroz)
reg_logit = glm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) + age +
                                kidslt6 + kidsge6),
     mroz, Binomial(), LogitLink())
reg_probit = glm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) + age +
                                 kidslt6 + kidsge6),
     mroz, Binomial(), ProbitLink())

# predictions for two "extreme" women:
X_new = DataFrame(nwifeinc=[100, 0], educ=[5, 17],
     exper=[0, 30], age=[20, 52],
     kidslt6=[2, 0], kidsge6=[0, 0])
predictions_lin = round.(predict(reg_lin, X_new), digits=5)
predictions_logit = round.(predict(reg_logit, X_new), digits=5)
predictions_probit = round.(predict(reg_probit, X_new), digits=5)

println("predictions_lin = $predictions_lin\n")
println("predictions_logit = $predictions_logit\n")
println("predictions_probit = $predictions_probit")
```

Output of Script 17.6: **Example-17-1-6.jl**

```
predictions_lin = [-0.41046, 1.04281]

predictions_logit = [0.00522, 0.95005]

predictions_probit = [0.00107, 0.95987]
```

Figure 17.1. Predictions from Binary Response Models (Simulated Data)



If we only have one regressor, predicted values can nicely be plotted against it. Figure 17.1 shows such a figure for a simulated data set. For interested readers, the script used for generating the data and the figure is printed as Script 17.7 (`Binary-Predictions.jl`) in Appendix IV (p. 374). In this example, the linear probability model clearly predicts probabilities outside of the "legal" area between 0 and 1. The logit and probit models yield almost identical predictions. This is a general finding that holds for most data sets.

17.1.5. Partial Effects

The parameters of linear regression models have straightforward interpretations: β_j measures the *ceteris paribus* effect of x_j on $\mathrm{E}(y|\mathbf{x})$. The parameters of nonlinear models like logit and probit have a less straightforward interpretation since the linear index $\mathbf{x}\boldsymbol{\beta}$ affects $\hat{y}$ through the link function G.

A useful measure of the influence is the partial effect (or marginal effect) which in a graph like Figure 17.1 is the slope and has the same interpretation as the parameters in the linear model. Because of the chain rule, it is

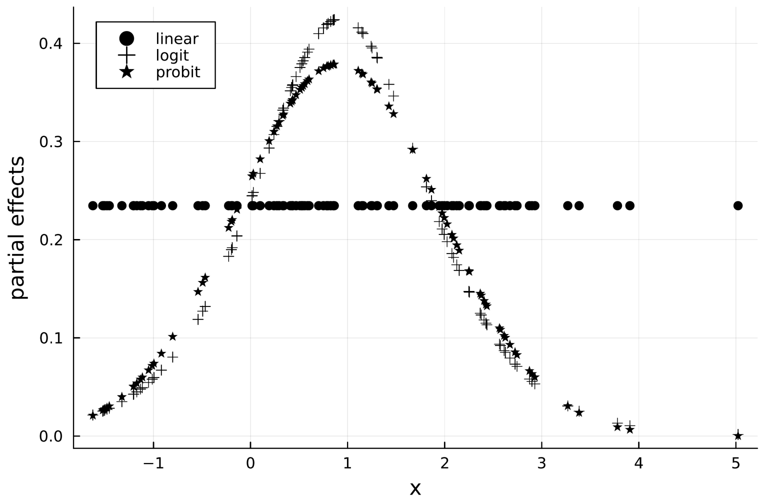
$$\frac{\partial \hat{y}}{\partial x_j} = \frac{\partial G(\hat{\beta}_0 + \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_k x_k)}{\partial x_j} \tag{17.9}$$

$$= \hat{\beta}_j \cdot g(\hat{\beta}_0 + \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_k x_k), \tag{17.10}$$

where $g(z)$ is the derivative of the link function $G(z)$. So

- for the probit model, the partial effect is
  $\frac{\partial \hat{y}}{\partial x_j} = \hat{\beta}_j \cdot \phi(\mathbf{x}\hat{\boldsymbol{\beta}})$
- for the logit model, it is
  $\frac{\partial \hat{y}}{\partial x_j} = \hat{\beta}_j \cdot \lambda(\mathbf{x}\hat{\boldsymbol{\beta}})$

Figure 17.2. Partial Effects for Binary Response Models (Simulated Data)



where $\phi(z)$ and $\lambda(z)$ are the PDFs of the standard normal and the logistic distribution, respectively.

The partial effect depends on the value of $\mathbf{x}\hat{\boldsymbol{\beta}}$. The PDFs have the famous bell-shape with highest values in the middle and values close to zero in the tails. This is already obvious from Figure 17.1. Depending on the value of x, the slope of the probability differs. For our simulated data set, Figure 17.2 shows the estimated partial effects for all 100 observed x values. Interested readers can see the complete code for this as Script 17.8 (`Binary-Margeff.jl`) in Appendix IV (p. 375).

The fact that the partial effects differ by regressor values makes it harder to present the results in a concise and meaningful way. There are two common ways to aggregate the partial effects:

- Partial effects at the average: $PEA = \hat{\beta}_j \cdot g(\bar{\mathbf{x}}\hat{\boldsymbol{\beta}})$
- Average partial effects: $APE = \frac{1}{n}\sum_{i=1}^{n} \hat{\beta}_j \cdot g(\mathbf{x}_i\hat{\boldsymbol{\beta}}) = \hat{\beta}_j \cdot \overline{g(\mathbf{x}\hat{\boldsymbol{\beta}})}$

where $\bar{\mathbf{x}}$ is the vector of sample averages of the regressors and $\overline{g(\mathbf{x}\hat{\boldsymbol{\beta}})}$ is the sample average of g evaluated at the individual linear index $\mathbf{x}_i\hat{\boldsymbol{\beta}}$. Both measures multiply each coefficient $\hat{\beta}_j$ with a constant factor.

The first part of Script 17.9 (`Example-17-1-7.jl`) implements the APE calculations for our labor force participation example using already known functions:

1. The linear indices $\mathbf{x}_i\hat{\boldsymbol{\beta}}$ are calculated by using the regressor matrix in **`reg.mm.m`** and the point estimates obtained with **`coef`**.
2. The factors $\overline{g(\mathbf{x}\hat{\boldsymbol{\beta}})}$ are calculated by using the PDF functions **`pdf.(Logistic(), xb_logit)`** and **`pdf.(Normal(), xb_probit)`** from the **`Distributions`** package and then averaging over the sample with **`mean`**.
3. The APEs are calculated by multiplying the coefficients with the corresponding factor. Note that for the linear probability model, the partial effects are constant and simply equal to the coefficients.

The APEs for all variables (except the constant) don't differ too much between the models. Note that APEs for the constant do not have a direct meaningful interpretation. As a general observation, as long as we are interested in APEs only and not in individual predictions or partial effects and as long as not too many probabilities are close to 0 or 1, the linear probability model often works well enough.

Script 17.9: **Example-17-1-7.jl**

```
using WooldridgeDatasets, GLM, DataFrames, Statistics,
    Distributions, LinearAlgebra

mroz = DataFrame(wooldridge("mroz"))

# estimate models:
reg_lin = lm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) + age +
                             kidslt6 + kidsge6), mroz)
reg_logit = glm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) + age +
                                kidslt6 + kidsge6),
    mroz, Binomial(), LogitLink())
reg_probit = glm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) + age +
                                 kidslt6 + kidsge6),
    mroz, Binomial(), ProbitLink())

# average partial effects:
APE_lin = coef(reg_lin)

coefs_logit = coef(reg_logit)
xb_logit = reg_logit.mm.m * coefs_logit
factor_logit = mean(pdf.(Logistic(), xb_logit))
APE_logit = coefs_logit * factor_logit

coefs_probit = coef(reg_probit)
xb_probit = reg_probit.mm.m * coefs_probit
factor_probit = mean(pdf.(Normal(), xb_probit))
APE_probit = coefs_probit * factor_probit

# print results:
table_manual = DataFrame(
    coef_names=coeftable(reg_lin).rownms,
    APE_lin=round.(APE_lin, digits=5),
    APE_logit=round.(APE_logit, digits=5),
    APE_probit=round.(APE_probit, digits=5))
println("table_manual: \n$table_manual")
```

Output of Script 17.9: **Example-17-1-7.jl**

```
table_manual:
8×4 DataFrame
 Row | coef_names   APE_lin   APE_logit  APE_probit
     | String       Float64   Float64    Float64
     ------------------------------------------------
   1 | (Intercept)   0.58552    0.07598     0.08123
   2 | nwifeinc     -0.00341   -0.00381    -0.00362
   3 | educ          0.038      0.0395      0.03937
   4 | exper         0.03949    0.03676     0.0371
   5 | exper ^ 2    -0.0006    -0.00056    -0.00057
   6 | age          -0.01609   -0.01572    -0.0159
   7 | kidslt6      -0.26181   -0.25775    -0.26115
   8 | kidsge6       0.01301    0.01073     0.01083
```

17.2. Count Data: The Poisson Regression Model

Instead of just 0/1-coded binary data, count data can take any non-negative integer $0, 1, 2, \ldots$. If they take very large numbers (like the number of students in a school), they can be approximated reasonably well as continuous variables in linear models and estimated using OLS. If the numbers are relatively small (like the number of children of a mother), this approximation might not work well. For example, predicted values can become negative.

The Poisson regression model is the most basic and convenient model explicitly designed for count data. The probability that y takes any value $h \in \{0, 1, 2, \ldots\}$ for this model can be written as

$$P(y = h|\mathbf{x}) = \frac{e^{-e^{\mathbf{x}\boldsymbol{\beta}}} \cdot e^{h \cdot \mathbf{x}\boldsymbol{\beta}}}{h!}. \tag{17.11}$$

The parameters of the Poisson model are much easier to interpret than those of a probit or logit model. In this model, the conditional mean of y is

$$E(y|\mathbf{x}) = e^{\mathbf{x}\boldsymbol{\beta}}, \tag{17.12}$$

so each slope parameter β_j has the interpretation of a semi elasticity:

$$\frac{\partial E(y|\mathbf{x})}{\partial x_j} = \beta_j \cdot e^{\mathbf{x}\boldsymbol{\beta}} = \beta_j \cdot E(y|\mathbf{x}) \tag{17.13}$$

$$\Leftrightarrow \beta_j = \frac{1}{E(y|\mathbf{x})} \cdot \frac{\partial E(y|\mathbf{x})}{\partial x_j}. \tag{17.14}$$

If x_j increases by one unit (and the other regressors remain the same), $E(y|\mathbf{x})$ will increase roughly by $100 \cdot \beta_j$ percent (the exact value is once again $100 \cdot (e^{\beta_j} - 1)$).

A problem with the Poisson model is that it is quite restrictive. The Poisson distribution implicitly restricts the variance of y to be equal to its mean. If this assumption is violated but the conditional mean is still correctly specified, the Poisson parameter estimates are consistent, but the standard errors and all inferences based on them are invalid. A solution is to interpret the Poisson estimators as quasi-maximum likelihood estimators (QMLE). Similar to the heteroscedasticity-robust inference for OLS discussed in Section 8.1, the standard errors can be adjusted.

Estimating Poisson regression models in **`GLM`** is straightforward. Given the data set **`sample`** contains variables **`y`**, **`x1`**, **`x2`**, **`x3`**, with the respective data of our sample, we can estimate the model with the following code:

```
reg_poisson = glm(@formula(y ~ x1 + x2 + x3), sample, Poisson())
```

For the more robust QMLE standard errors, we use the procedure described in Wooldridge (2019, Chapter 17.3) and adjust the standard errors in **`reg_poisson`**.

Wooldridge, Example 17.3: Poisson Regression for Number of Arrests

We apply the Poisson regression model to study the number of arrests of young men in 1986. Script 17.10 (`Example-17-3.jl`) imports the data and first estimates a linear regression model using OLS. Then, a Poisson model is estimated using **`Poisson`**. Finally, we adjust the standard errors for a potential violation of the Poisson distribution using the QMLE specification. By construction, the parameter estimates are the same, but the standard errors are larger.

Script 17.10: `Example-17-3.jl`

```
using WooldridgeDatasets, GLM, DataFrames

crime1 = DataFrame(wooldridge("crime1"))

# estimate linear model:
reg_lin = lm(@formula(narr86 ~ pcnv + avgsen + tottime + ptime86 + qemp86 +
                               inc86 + black + hispan + born60), crime1)
table_lin = coeftable(reg_lin)
println("table_lin: \n$table_lin\n")

# estimate Poisson model:
reg_poisson = glm(@formula(narr86 ~ pcnv + avgsen + tottime + ptime86 + qemp86 +
                                    inc86 + black + hispan + born60),
    crime1, Poisson())
table_poisson = coeftable(reg_poisson)
println("table_poisson: \n$table_poisson\n")

# estimate Quasi-Poisson model:
yhat = predict(reg_poisson)
resid = crime1.narr86 .- yhat
sigma_sq = 1 / (2725 - 9 - 1) * sum(resid .^ 2 ./ yhat)
table_qpoisson = coeftable(reg_poisson)
table_qpoisson.cols[2] = table_qpoisson.cols[2] * sqrt(sigma_sq)
println("table_qpoisson: \n$table_qpoisson")
```

Output of Script 17.10: `Example-17-3.jl`

```
table_lin:

                  Coef.   Std. Error      t  Pr(>|t|)    Lower 95%     Upper 95%

(Intercept)   0.576566   0.0378945    15.22    <1e-49   0.502261     0.650871
pcnv         -0.131886   0.0404037    -3.26    0.0011  -0.211111    -0.0526609
avgsen       -0.0113316  0.0122413    -0.93    0.3547  -0.0353348    0.0126717
tottime       0.0120693  0.00943641    1.28    0.2010  -0.00643401   0.0305725
ptime86      -0.0408735  0.00881303   -4.64    <1e-05  -0.0581544   -0.0235925
qemp86       -0.0513099  0.0144862    -3.54    0.0004  -0.079715    -0.0229047
inc86        -0.0014617  0.000343021  -4.26    <1e-04  -0.00213431  -0.000789092
black         0.32701    0.0454264     7.20    <1e-12   0.237936     0.416083
hispan        0.193809   0.0397156     4.88    <1e-05   0.115933     0.271685
born60       -0.022465   0.0332945    -0.67    0.4999  -0.0877502    0.0428202


table_poisson:

                  Coef.  Std. Error      z  Pr(>|z|)    Lower 95%    Upper 95%

(Intercept)  -0.599589   0.067229    -8.92    <1e-18  -0.731355    -0.467822
pcnv         -0.401571   0.0849386   -4.73    <1e-05  -0.568047    -0.235094
avgsen       -0.0237723  0.0199427   -1.19    0.2332  -0.0628592    0.0153146
tottime       0.0244904  0.0147467    1.66    0.0968  -0.00441267   0.0533934
ptime86      -0.0985584  0.0206858   -4.76    <1e-05  -0.139102    -0.058015
qemp86       -0.0380188  0.0290172   -1.31    0.1901  -0.0948916    0.0188539
inc86        -0.0080807  0.00104053  -7.77    <1e-14  -0.0101201   -0.00604129
black         0.660837   0.0738187    8.95    <1e-18   0.516155     0.805519
hispan        0.499813   0.073907     6.76    <1e-10   0.354958     0.644668
born60       -0.0510287  0.064036    -0.80    0.4255  -0.176537     0.0744795
```

```
table_qpoisson:

                 Coef.  Std. Error      z  Pr(>|z|)    Lower 95%    Upper 95%

(Intercept)  -0.599589    0.0827979  -8.92    <1e-18  -0.731355    -0.467822
pcnv         -0.401571    0.104609   -4.73    <1e-05  -0.568047    -0.235094
avgsen       -0.0237723   0.024561   -1.19    0.2332  -0.0628592    0.0153146
tottime       0.0244904   0.0181617   1.66    0.0968  -0.00441267   0.0533934
ptime86      -0.0985584   0.0254762  -4.76    <1e-05  -0.139102    -0.058015
qemp86       -0.0380188   0.035737   -1.31    0.1901  -0.0948916    0.0188539
inc86        -0.0080807   0.0012815  -7.77    <1e-14  -0.0101201   -0.00604129
black         0.660837    0.0909135   8.95    <1e-18   0.516155     0.805519
hispan        0.499813    0.0910224   6.76    <1e-10   0.354958     0.644668
born60       -0.0510287   0.0788654  -0.80    0.4255  -0.176537     0.0744795
```

17.3. Corner Solution Responses: The Tobit Model

Corner solutions describe situations where the variable of interest is continuous but restricted in range. Typically, it cannot be negative. A significant share of people buy exactly zero amounts of alcohol, tobacco, or diapers. The Tobit model explicitly models dependent variables like this. It can be formulated in terms of a latent variable y^* that can take all real values. For it, the classical linear regression model assumptions MLR.1–MLR.6 are assumed to hold. If y^* is positive, we observe $y = y^*$. Otherwise, $y = 0$. Wooldridge (2019, Section 17.2) shows how to derive properties and the likelihood function for this model.

The problem of interpreting the parameters is similar to logit or probit models. While β_j measures the *ceteris paribus* effect of x_j on $\mathrm{E}(y^*|\mathbf{x})$, the interest is typically in y instead. The partial effect of interest can be written as

$$\frac{\partial \mathrm{E}(y|\mathbf{x})}{\partial x_j} = \beta_j \cdot \Phi\left(\frac{\mathbf{x}\boldsymbol{\beta}}{\sigma}\right) \tag{17.15}$$

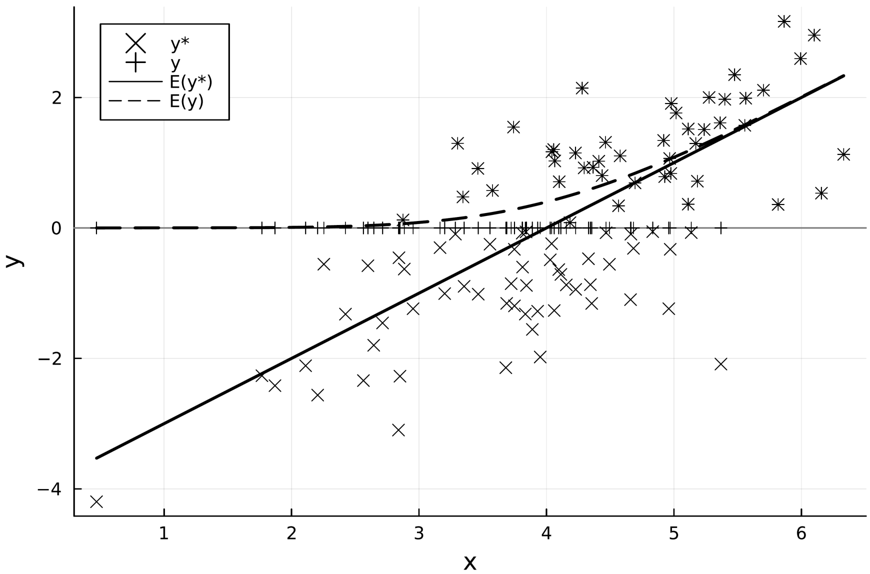
and again depends on the regressor values $\mathbf{x}$. To aggregate them over the sample, we can either calculate the partial effects at the average (PEA) or the average partial effect (APE) just like with the binary variable models.

Figure 17.3 depicts these properties for a simulated data set with only one regressor. Whenever $y^* > 0$, $y = y^*$ and the symbols $\times$ and $+$ are on top of each other. If $y^* < 0$, then $y = 0$. Therefore, the slope of $\mathrm{E}(y|x)$ gets close to zero for very low x values. The code that generated the data set and the graph is hidden as Script 17.11 (`Tobit-CondMean.jl`) in Appendix IV (p. 376).

Since there is no boxed routine of the Tobit model in *Julia*, we implement our own estimator. To do this, we have to come up with our own definition of a log likelihood. The function **`ll_tobit`** in Script 17.12 (`Example-17-2.jl`) uses the definition of the log likelihood in Wooldridge (2019). To keep things simple, we make no use of formula syntax and provide the data as matrices **`X`** and **`y`**. The function **`optim`** from the package **`Optim`** finds a minimum of the provided negative log likelihood, i.e. a maximum of the log likelihood.[1] We provide OLS results as a start solution for this optimization procedure. We finally print the estimated coefficients with **`Optim.minimizer(optimum)`** and the respective log likelihood with **`-optimum.minimum`**.

[1]For more information about the package, see Mogensen, Riseth, White, Holy, and other contributors (2017).

Figure 17.3. Conditional Means for the Tobit Model



Wooldridge, Example 17.2: Married Women's Annual Labor Supply

We have already estimated labor supply models for the women in the data set `mroz`, ignoring the fact that the hours worked is necessarily non-negative. Script 17.12 (`Example-17-2.jl`) estimates a Tobit model accounting for this fact.

Script 17.12: **Example-17-2.jl**

```
using WooldridgeDatasets, GLM, DataFrames, Statistics, Distributions,
    LinearAlgebra, Optim
include("../03/getMats.jl")

# load data and build data matrices:
mroz = DataFrame(wooldridge("mroz"))
f = @formula(hours ~ 1 + nwifeinc + educ + exper +
                     (exper^2) + age + kidslt6 + kidsge6)
xy = getMats(f, mroz)
y = xy[1]
X = xy[2]

# define a function that returns the negative log likelihood per observation
# (for details on the implementation see Wooldridge (2019), formula 17.22):
function ll_tobit(params, y, X)
    p = size(X, 2)
    beta = params[1:p]
    sigma = exp(params[p+1])
    y_hat = X * beta
    y_eq = (y .== 0)
    y_g = (y .> 0)
    ll = zeros(length(y))
    ll[y_eq] = log.(cdf.(Normal(), -y_hat[y_eq] / sigma))
    ll[y_g] = log.(pdf.(Normal(), (y.-y_hat)[y_g] / sigma)) .- log(sigma)
    # return the negative sum of log likelihoods for each observation:
    return -sum(ll)
end


# generate starting solution:
reg_ols = lm(@formula(hours ~ nwifeinc + educ + exper + (exper^2) +
                               age + kidslt6 + kidsge6), mroz)
resid_ols = residuals(reg_ols)
sigma_start = log(sum(resid_ols .^ 2) / length(resid_ols))
params_start = vcat(coef(reg_ols), sigma_start)

# maximize the log likelihood = minimize the negative of the log likelihood:
optimum = optimize(par -> ll_tobit(par, y, X), params_start, Newton())
mle_est = Optim.minimizer(optimum)
ll = -optimum.minimum

# print results:
table_mle = DataFrame(
    coef_names=vcat(coeftable(reg_ols).rownms, "exp_sigma"),
    mle_est=round.(mle_est, digits=5))
println("table_mle: \n$table_mle\n")
println("ll = $ll")
```

Output of Script 17.12: `Example-17-2.jl`

```
table_mle:
9×2 DataFrame
 Row | coef_names   mle_est
     | String       Float64
     ---------------------------------------------
   1 | (Intercept)   965.305
   2 | nwifeinc       -8.81424
   3 | educ           80.6456
   4 | exper         131.564
   5 | exper ^ 2      -1.86416
   6 | age           -54.405
   7 | kidslt6      -894.022
   8 | kidsge6       -16.218
   9 | exp_sigma        7.02289

ll = -3819.094558766155
```

17.4. Censored and Truncated Regression Models

Censored regression models are closely related to Tobit models. In fact, their parameters can be estimated with nearly the same procedure discussed in the previous section. General censored regression models also start from a latent variable y^*. The observed dependent variable y is equal to y^* for some (the uncensored) observations. For the other observations, we only know an upper or lower bound for y^*. In the basic Tobit model, we observe $y = y^*$ in the "uncensored" cases with $y^* > 0$ and we only know that $y^* \leq 0$ if we observe $y = 0$. The censoring rules can be much more general. There could be censoring from above or the thresholds can vary from observation to observation.

The main difference between Tobit and censored regression models is the interpretation. In the former case, we are interested in the observed y, in the latter case, we are interested in the underlying y^*.[2] Censoring is merely a data problem that has to be accounted for instead of a logical feature of the dependent variable. We already know how to estimate Tobit models. With censored regression, we can use the same tools. The problem of calculating partial effects does not exist in this case since we are interested in the linear $\mathrm{E}(y^*|\mathbf{x})$ and the slope parameters are directly equal to the partial effects of interest.

Wooldridge, Example 17.4: Duration of Recidivism

We are interested in the criminal prognosis of individuals released from prison. We model the time it takes them to be arrested again. Explanatory variables include demographic characteristics as well as a dummy variable **`workprg`** indicating the participation in a work program during their time in prison. The 1445 former inmates observed in the data set `recid` were followed for a while.

During that time, 893 inmates were not arrested again. For them, we only know that their true duration y^* is at least **`durat`**, which for them is the time between the release and the end of the observation period, so we have right censoring. The threshold of censoring differs by individual depending on when they were released.

In Script 17.13 (`Example-17-4.jl`) we implement the log likelihood optimization similar to Script 17.12 (`Example-17-2.jl`). Because of the more complicated selection rule, we have to add a parameter **`cens`**, which is a dummy variable indicating *censored* observations. Details on the foundation of the implementation for the log likelihood with right censored data is provided in Wooldridge (2019).

[2] Wooldridge (2019, Section 17.4) uses the notation w instead of y and y instead of y^*.

Estimates can directly be interpreted. Because of the logarithmic specification, they represent semi-elasticities. For example, do married individuals take around $100 \cdot \hat{\beta} = 34\%$ longer to be arrested again. (Actually, the accurate number is $100 \cdot (e^{\hat{\beta}} - 1) = 40\%$.)

Script 17.13: Example-17-4.jl

```
using WooldridgeDatasets, GLM, DataFrames, Statistics, Distributions,
        LinearAlgebra, Optim

# load data and build data matrices:
recid = DataFrame(wooldridge("recid"))
f = @formula(ldurat ~ 1 + workprg + priors + tserved +
                      felon + alcohol + drugs + black +
                      married + educ + age)
xy = getMats(f, recid)
y = xy[1]
X = xy[2]

# define dummy for censored observations:
censored = recid.cens .!= 0

# generate starting solution:
reg_ols = lm(@formula(ldurat ~ workprg + priors + tserved +
                               felon + alcohol + drugs +
                               black + married +
                               educ + age), recid)
resid_ols = residuals(reg_ols)
sigma_start = log(sum(resid_ols .^ 2) / length(resid_ols))
params_start = vcat(coef(reg_ols), sigma_start)

# define a function that returns the negative log likelihood per observation:
function ll_censreg(params, y, X, cens)
        p = size(X, 2)
        beta = params[1:p]
        sigma = exp(params[p+1])
        y_hat = X * beta
        ll = zeros(length(y))
        # uncensored:
        ll[.!cens] = log.(pdf.(Normal(),
                (y.-y_hat)[.!cens] / sigma)) .- log(sigma)
        # censored:
        ll[cens] = log.(cdf.(Normal(), -(y.-y_hat)[cens] / sigma))

        # return the negative sum of log likelihoods for each observation:
        return -sum(ll)
end

# maximize the log likelihood = minimize the negative of the log likelihood:
optimum = optimize(par -> ll_censreg(par, y, X, censored), params_start, Newton())
mle_est = Optim.minimizer(optimum)
ll = -optimum.minimum

# print results of MLE:
table_mle = DataFrame(
        coef_names=vcat(coeftable(reg_ols).rownms, "exp_sigma"),
        mle_est=round.(mle_est, digits=5))
println("table_mle: \n$table_mle\n")
println("ll = $ll")
```

Output of Script 17.13: `Example-17-4.jl`

```
table_mle:
12×2 DataFrame
 Row | coef_names   mle_est
     | String       Float64
     -------------------------------------------
   1 | (Intercept)   4.09938
   2 | workprg      -0.06257
   3 | priors       -0.13725
   4 | tserved      -0.01933
   5 | felon         0.44399
   6 | alcohol      -0.63491
   7 | drugs        -0.29816
   8 | black        -0.54272
   9 | married       0.34068
  10 | educ          0.02292
  11 | age           0.00391
  12 | exp_sigma     0.59359

ll = -1597.058962306061
```

Truncation is a more serious problem than censoring since our observations are more severely affected. If the true latent variable y^* is above or below a certain threshold, the individual is not even sampled. We therefore do not even have any information. Classical truncated regression models rely on parametric and distributional assumptions to correct this problem. In *Julia* they can be implemented by providing an adjusted log likelihood just as discussed above. We will not go into details here, but Wooldridge (2019) describes how to implement the log likelihood.

Figure 17.4 shows results for a simulated data set. Because it is simulated, we actually know the values for everybody (hollow and solid dots). In our sample, we only observe those with $y > 0$ (solid dots). When applying OLS to this sample, we get a downward biased slope (dashed line). Truncated regression fixes this problem and gives a consistent slope estimator (solid line). Script 17.14 (`TruncReg-Simulation.jl`) which generated the data set and the graph is shown in Appendix IV (p. 379).

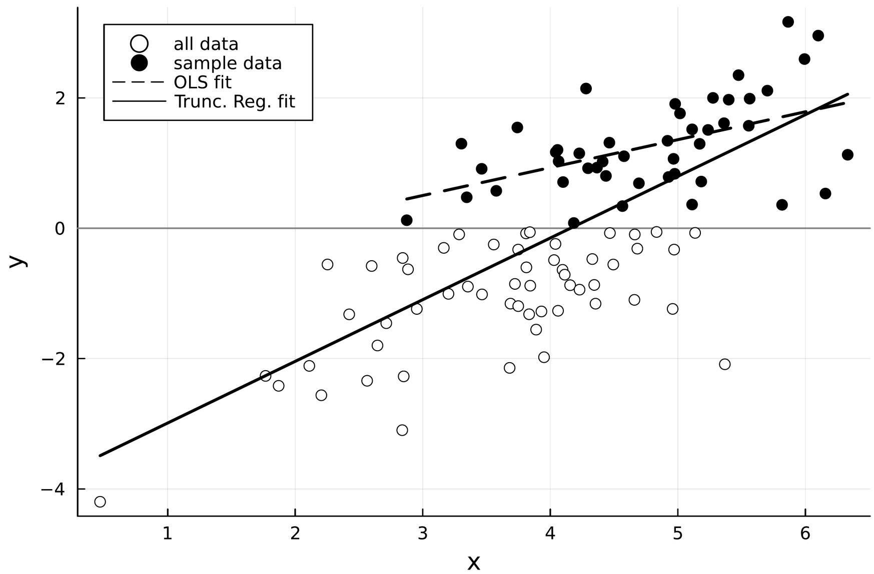
17.5. Sample Selection Corrections

Sample selection models are related to truncated regression models. We do have a random sample from the population of interest, but we do not observe the dependent variable y for a non-random sub-sample. The sample selection is not based on a threshold for y but on some other selection mechanism.

Heckman's selection model consists of a probit-like model for the binary fact whether y is observed and a linear regression-like model for y. Selection can be driven by the same determinants as y but should have at least one additional factor excluded from the equation for y. Wooldridge (2019, Section 17.5) discusses the specification and estimation of these models in more detail.

The classical Heckman selection model can be estimated in two steps using software for probit and OLS as discussed by Wooldridge (2019). We will demonstrate this two-step approach with **`GLM`**.

Figure 17.4. Truncated Regression: Simulated Example



Wooldridge, Example 17.5: Wage offer Equation for Married Women

We once again look at the sample of women in the data set MROZ. Of the 753 women, 428 worked (inlf=1) and the rest did not work (inlf=0). For the latter, we do not observe the wage they would have gotten had they worked. Script 17.15 (Example-17-5.jl) estimates the Heckman selection model using two formulas: one for the selection and one for the wage equation.

Script 17.15: Example-17-5.jl

```
using WooldridgeDatasets, GLM, DataFrames, Distributions

# load data and build data matrices:
mroz = DataFrame(wooldridge("mroz"))

# step 1 (use all n observations to estimate a probit model of s_i on z_i):
reg_probit = glm(@formula(inlf ~ educ + exper +
                                 (exper^2) + nwifeinc +
                                 age + kidslt6 + kidsge6),
    mroz, Binomial(), ProbitLink())
pred_inlf_linpart = quantile.(Normal(), fitted(reg_probit))
mroz.inv_mills = pdf.(Normal(), pred_inlf_linpart) ./
                 cdf.(Normal(), pred_inlf_linpart)

# step 2 (regress y_i on x_i and inv_mills in sample selection):
mroz_subset = subset(mroz, :inlf => ByRow(==(1)))
reg_heckit = lm(@formula(lwage ~ educ + exper + (exper^2) +
                                 inv_mills), mroz_subset)

# print results:
table_reg_heckit = coeftable(reg_heckit)
println("table_reg_heckit: \n$table_reg_heckit")
```

Output of Script 17.15: Example-17-5.jl

```
table_reg_heckit:

                    Coef.   Std. Error      t  Pr(>|t|)    Lower 95%   Upper 95%

(Intercept)  -0.578102     0.306723     -1.88    0.0601  -1.18099     0.0247893
educ          0.109065     0.0156096     6.99    <1e-10   0.0783835   0.139748
exper         0.0438873    0.0163534     2.68    0.0076   0.0117433   0.0760313
exper ^ 2    -0.000859113  0.000441396  -1.95    0.0523  -0.00172672  8.48967e-6
inv_mills     0.0322614    0.134388      0.24    0.8104  -0.23189     0.296413
```

18. Advanced Time Series Topics

After we have introduced time series concepts in Chapters 10 – 12, this chapter touches on some more advanced topics in time series econometrics. Namely, we look at infinite distributed lag models in Section 18.1, unit roots tests in Section 18.2, spurious regression in Section 18.3, cointegration in Section 18.4 and forecasting in Section 18.5.

18.1. Infinite Distributed Lag Models

We have covered finite distributed lag models in Section 10.3. We have estimated those and related models in *Julia* using the package **GLM**. In *infinite* distributed lag models, shocks in the regressors z_t have an infinitely long impact on $y_t, y_{t+1}, \dots$. The long-run propensity is the overall future effect of increasing z_t by one unit and keeping it at that level.

Without further restrictions, infinite distributed lag models cannot be estimated. Wooldridge (2019, Section 18.1) discusses two different models. The **geometric (or Koyck)** distributed lag model boils down to a linear regression equation in terms of lagged dependent variables

$$y_t = \alpha_0 + \gamma z_t + \rho y_{t-1} + v_t \tag{18.1}$$

and has a long-run propensity of

$$LRP = \frac{\gamma}{1-\rho}. \tag{18.2}$$

The **rational** distributed lag model can be written as a somewhat more general equation

$$y_t = \alpha_0 + \gamma_0 z_t + \rho y_{t-1} + \gamma_1 z_{t-1} + v_t \tag{18.3}$$

and has a long-run propensity of

$$LRP = \frac{\gamma_0 + \gamma_1}{1-\rho}. \tag{18.4}$$

In terms of the implementation of these models, there is nothing really new compared to Section 10.3. The only difference is that we include lagged dependent variables as regressors.

Wooldridge, Example 18.1: Housing Investment and Residential Price Inflation

Script 18.1 (`Example-18-1.jl`) implements the geometric and the rational distributed lag models for the housing investment equation. The dependent variable is detrended by simply using the residual of a regression on a linear time trend. We store this detrended variable in the data frame.
The two models are estimated using the **`lm`** function and a regression table very similar to Wooldridge (2019, Table 18.1) is produced. Finally, we estimate the LRP for both models using the formulas given above. We extract the respective coefficients and do the calculations. For example, **`coef(reg_koyck)[2]`** is the coefficient for γ in the geometric distributed lag model.

Script 18.1: `Example-18-1.jl`

```
using WooldridgeDatasets, GLM, DataFrames

hseinv = DataFrame(wooldridge("hseinv"))

# add lags and detrend:
reg_trend = lm(@formula(linvpc ~ t), hseinv)
hseinv.linvpc_det = residuals(reg_trend)
hseinv.gprice_lag1 = lag(hseinv.gprice, 1)
hseinv.linvpc_det_lag1 = lag(hseinv.linvpc_det, 1)

# Koyck geometric d.l.:
reg_koyck = lm(@formula(linvpc_det ~ gprice +
                                     linvpc_det_lag1), hseinv)
table_koyck = coeftable(reg_koyck)
println("table_koyck: \n$table_koyck\n")

# rational d.l.:
reg_rational = lm(@formula(linvpc_det ~ gprice + linvpc_det_lag1 +
                                        gprice_lag1), hseinv)
table_rational = coeftable(reg_rational)
println("table_rational: \n$table_rational\n")

# calculate LRP as...
# gprice / (1 - linvpc_det_lag1):
lrp_koyck = coef(reg_koyck)[2] / (1 - coef(reg_koyck)[3])
println("lrp_koyck = $lrp_koyck\n")

# and (gprice + gprice_lag1) / (1 - linvpc_det_lag1):
lrp_rational = (coef(reg_rational)[2] + coef(reg_rational)[4]) /
               (1 - coef(reg_rational)[3])
println("lrp_rational = $lrp_rational")
```

Output of Script 18.1: `Example-18-1.jl`

```
table_koyck:
                       Coef.  Std. Error      t  Pr(>|t|)   Lower 95%  Upper 95%

(Intercept)      -0.00996294    0.017916  -0.56    0.5814  -0.046232   0.0263062
gprice            3.09483       0.933327   3.32    0.0020   1.20541    4.98425
linvpc_det_lag1   0.339901      0.131588   2.58    0.0138   0.0735153  0.606288


table_rational:
                       Coef.  Std. Error      t  Pr(>|t|)   Lower 95%   Upper 95%

(Intercept)       0.00586852   0.0169326   0.35    0.7309  -0.0284725   0.0402095
gprice            3.25635      0.970323    3.36    0.0019   1.28845     5.22426
linvpc_det_lag1   0.547171     0.151671    3.61    0.0009   0.239567    0.854774
gprice_lag1      -2.93634      0.973186   -3.02    0.0047  -4.91006    -0.962632


lrp_koyck = 4.688434194769012

lrp_rational = 0.7066808046888278
```

18.2. Testing for Unit Roots

We have covered strongly dependent unit root processes in Chapter 11 and promised to supply tests for unit roots later. There are several tests available. Conceptually, the Dickey-Fuller (DF) test is the simplest. If we want to test whether variable y has a unit root, we regress Δy_t on y_{t-1}. The test statistic is the usual t test statistic of the slope coefficient. One problem is that because of the unit root, this test statistic is *not* t or normally distributed, not even asymptotically. Instead, we have to use special distribution tables for the critical values. The distribution also depends on whether we allow for a time trend in this regression.

The augmented Dickey-Fuller (ADF) test is a generalization that allows for richer dynamics in the process of y. To implement it, we add lagged values $\Delta y_{t-1}, \Delta y_{t-2}, \ldots$ to the differenced regression equation.

Of course, working with the special (A)DF tables of critical values is somewhat inconvenient. The package **`HypothesisTests`** offers automated DF and ADF tests for models with time trends. The command **`ADFTest(y, :trend, lag)`** performs an ADF test with selecting the number of lags in Δy as the integer **`lag`**. The argument **`:trend`** includes a time trend and other options are available. For example, **`ADFTest(y, :none, 0)`** requests zero lags without a constant and time trend, i.e. a simple DF test.

Wooldridge, Example 18.4: Unit Root in Real GDP

Script 18.2 (`Example-18-4.jl`) implements an ADF test for the logarithm of U.S. real GDP including a linear time trend. For a test with one lag in Δy and time trend, the equation to estimate is

$$\Delta y = \alpha + \theta y_{t-1} + \gamma_1 \Delta y_{t-1} + \delta t + e_t.$$

We already know how to implement such a regression using **`lm`**, so we demonstrate the use of **`ADFTest`**. The relevant test statistic is $t = -2.42073$ and the critical values are also given in the output. More conveniently, the script also reports a p value of 0.3687. So the null hypothesis of a unit root cannot be rejected with any reasonable significance level.

Script 18.2: `Example-18-4.jl`

```
using WooldridgeDatasets, DataFrames, HypothesisTests

inven = DataFrame(wooldridge("inven"))
inven.lgdp = log.(inven.gdp)

# automated ADF:
adf_lag = 1
res_ADF_aut = ADFTest(inven.lgdp, :trend, adf_lag)
println("res_ADF_aut: \n$res_ADF_aut")
```

Output of Script 18.2: `Example-18-4.jl`

```
res_ADF_aut:
Augmented Dickey-Fuller unit root test
--------------------------------------
Population details:
    parameter of interest:   coefficient on lagged non-differenced variable
    value under h_0:         0
    point estimate:          -0.209621

Test summary:
    outcome with 95% confidence: fail to reject h_0
    p-value:                     0.3687

Details:
    sample size in regression:          35
    number of lags:                     1
    ADF statistic:                      -2.42073
    Critical values at 1%, 5%, and 10%: [-4.22686 -3.53665 -3.20024]
```

18.3. Spurious Regression

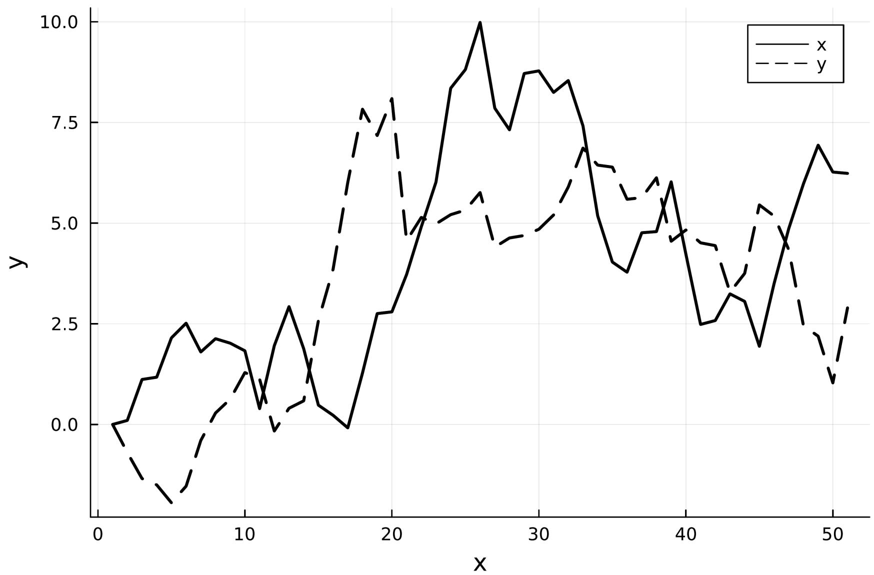
Unit roots generally destroy the usual (large sample) properties of estimators and tests. A leading example is spurious regression. Suppose two variables x and y are completely unrelated but both follow a random walk:

$$\begin{aligned} x_t &= x_{t-1} + a_t \\ y_t &= y_{t-1} + e_t, \end{aligned}$$

where a_t and e_t are i.i.d. random innovations. If we want to test whether they are related from a random sample, we could simply regress y on x. A t test should reject the (true) null hypothesis that the slope coefficient is equal to zero with a probability of α, for example 5%. The phenomenon of spurious regression implies that this happens much more often.

Script 18.3 (`Simulate-Spurious-Regression-1.jl`) simulates this model for one sample. Remember from Section 11.2 how to simulate a random walk in a simple way: with a starting value of zero, it is just the cumulative sum of the innovations. The time series for this simulated sample of size $n = 50$ is shown in Figure 18.1. When we regress y on x, the t statistic for the slope parameter is larger than 3 with a p value much smaller than 1%. So we would reject the (correct) null hypothesis that the variables are unrelated.

Figure 18.1. Spurious Regression: Simulated Data from Script 18.3



Script 18.3: `Simulate-Spurious-Regression-1.jl`

```
using Random, Distributions, Statistics, Plots, GLM, DataFrames

# set the random seed:
Random.seed!(12345)

# i.i.d. N(0,1) innovations:
n = 51
e = rand(Normal(), n)
e[1] = 0
a = rand(Normal(), n)
a[1] = 0

# independent random walks:
x = cumsum(a)
y = cumsum(e)
sim_data = DataFrame(y=y, x=x)

# regression:
reg = lm(@formula(y ~ x), sim_data)
reg_table = coeftable(reg)
println("reg_table: \n$reg_table")

# graph:
plot(x, color="black", linewidth=2, linestyle=:solid, label="x")
plot!(y, color="black", linewidth=2, linestyle=:dash, label="y")
ylabel!("y")
xlabel!("x")
savefig("JlGraphs/Simulate-Spurious-Regression-1.pdf")
```

Output of Script 18.3: Simulate-Spurious-Regression-1.jl

```
reg_table:

                Coef.   Std. Error     t  Pr(>|t|)  Lower 95%  Upper 95%

(Intercept)  1.75017     0.610701  2.87    0.0061   0.522923   2.97742
x            0.430938    0.123895  3.48    0.0011   0.181963   0.679914
```

We know that by definition, a valid test should reject a true null hypothesis with a probability of α, so maybe we were just unlucky with the specific sample we took. We therefore repeat the same analysis with 10,000 samples from the same data generating process in Script 18.4 (`Simulate-Spurious-Regression-2.jl`). For each of the samples, we store the p value of the slope parameter in a vector named **`pvals`**. After these simulations are run, we simply check how often we would have rejected $H_0 : \beta_1 = 0$ by comparing these p values with 0.05.

We find that in 6,721 of the samples, so in 67% instead of $\alpha = 5\%$, we rejected H_0. So the t test seriously screws up the statistical inference because of the unit roots.

Script 18.4: Simulate-Spurious-Regression-2.jl

```
using Random, Distributions, Statistics, Plots, GLM, DataFrames

# set the random seed:
Random.seed!(12345)

pvals = zeros(10000)

for i in 1:10000
    # i.i.d. N(0,1) innovations:
    n = 51
    e = rand(Normal(), n)
    e[1] = 0
    a = rand(Normal(), n)
    a[1] = 0

    # independent random walks:
    x = cumsum(a)
    y = cumsum(e)
    sim_data = DataFrame(y=y, x=x)

    # regression:
    reg = lm(@formula(y ~ x), sim_data)
    reg_table = coeftable(reg)

    # save the p value of x:
    pvals[i] = reg_table.cols[4][2]
end

# how often is p<=5%:
count_pval_smaller = sum(pvals .<= 0.05)  # counts true elements
println("count_pval_smaller = $count_pval_smaller\n")

# how often is p>5%:
count_pval_greater = sum(pvals .> 0.05)  # counts true elements
println("count_pval_greater = $count_pval_greater")
```

Output of Script 18.4: `Simulate-Spurious-Regression-2.jl`

```
count_pval_smaller = 6721

count_pval_greater = 3279
```

18.4. Cointegration and Error Correction Models

In Section 18.3, we just saw that it is not a good idea to do linear regression with integrated variables. This is not generally true. If two variables are not only integrated (i.e. they have a unit root), but *cointegrated*, linear regression with them can actually make sense. Often, economic theory suggests a stable long-run relationship between integrated variables which implies cointegration. Cointegration implies that in the regression equation

$$y_t = \beta_0 + \beta_1 x_t + u_t,$$

the error term u does not have a unit root, while both y and x do. A test for cointegration can be based on this finding: We first estimate this model by OLS and then test for a unit root in the residuals $\hat{u}$. Again, we have to adjust the distribution of the test statistic and critical values. This approach is called Engle-Granger test in Wooldridge (2019, Section 18.4) or Phillips–Ouliaris (PO) test. If we find cointegration, we can estimate error correction models. In the Engle-Granger procedure, these models can be estimated in a two-step procedure using OLS.

18.5. Forecasting

One major goal of time series analysis is forecasting. Given the information we have today, we want to give our best guess about the future and also quantify our uncertainty. Given a time series model for y, the best guess for y_{t+1} given information I_t is the conditional mean of $\mathrm{E}(y_{t+1}|I_t)$. For a model like

$$y_t = \delta_0 + \alpha_1 y_{t-1} + \gamma_1 z_{t-1} + u_t, \tag{18.5}$$

suppose we are at time t and know both y_t and z_t and want to predict y_{t+1}. Also suppose that $\mathrm{E}(u_t|I_{t-1}) = 0$. Then,

$$\mathrm{E}(y_{t+1}|I_t) = \delta_0 + \alpha_1 y_t + \gamma_1 z_t \tag{18.6}$$

and our prediction from an estimated model would be $\hat{y}_{t+1} = \hat{\delta}_0 + \hat{\alpha}_1 y_t + \hat{\gamma}_1 z_t$.

We already know how to get in-sample and (hypothetical) out-of-sample predictions including forecast intervals from linear models using the command **`predict`**. It can also be used for our purposes.

There are several ways how the performance of forecast models can be evaluated. It makes a lot of sense not to look at the model fit within the estimation sample but at the out-of-sample forecast performances. Suppose we have used observations $y_1, \dots, y_n$ for estimation and additionally have observations $y_{n+1}, \dots, y_{n+m}$. For this set of observations, we obtain out-of-sample forecasts $f_{n+1}, \dots, f_{n+m}$ and calculate the m forecast errors

$$e_t = y_t - f_t \qquad \text{for } t = n+1, \dots, n+m. \tag{18.7}$$

We want these forecast errors to be as small (in absolute value) as possible. Useful measures are the root mean squared error ($RMSE$) and the mean absolute error (MAE):

$$RMSE = \sqrt{\frac{1}{m} \sum_{h=1}^{m} e_{n+h}^2} \tag{18.8}$$

$$MAE = \frac{1}{m} \sum_{h=1}^{m} \left| e_{n+h} \right| \tag{18.9}$$

Wooldridge, Example 18.8: Forecasting the U.S. Unemployment Rate

Script 18.5 (`Example-18-8.jl`) estimates two simple models for forecasting the unemployment rate. The first one is a basic AR(1) model with only lagged unemployment as a regressor, the second one adds lagged inflation. We generate the data frame **`yt96`** with **`subset`** to restrict the estimation sample to years until 1996. After the estimation, we make predictions with **`predict`**.
Script 18.5 (`Example-18-8.jl`) also calculates the forecast errors of the unemployment rate for the two models used in Example 18.8. Predictions are made for the other seven available years until 2003. The actual unemployment rate and the forecasts are plotted – the result is shown in Figure 18.2. Finally, we calculate the $RMSE$ and MAE for both models. Both measures suggest that the second model including the lagged inflation performs better.

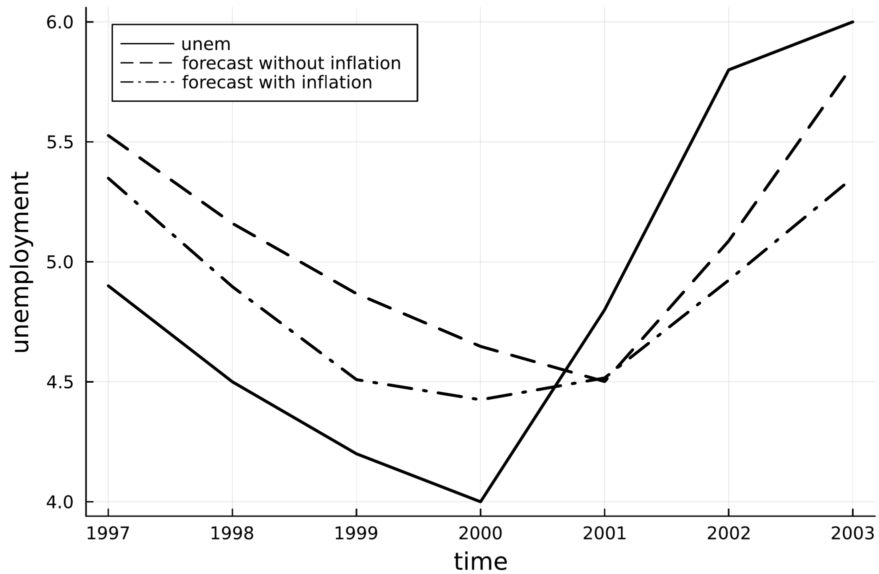
Script 18.5: **Example-18-8.jl**

```
using WooldridgeDatasets, GLM, DataFrames, Statistics, Plots

phillips = DataFrame(wooldridge("phillips"))

# estimate models:
yt96 = subset(phillips, :year => ByRow(<=(1996)))
reg_1 = lm(@formula(unem ~ unem_1), yt96)
reg_2 = lm(@formula(unem ~ unem_1 + inf_1), yt96)

# predictions for 1997-2003:
yf97 = subset(phillips, :year => ByRow(>(1996)))
pred_1 = round.(predict(reg_1, yf97), digits=5)
println("pred_1 = $pred_1\n")
pred_2 = round.(predict(reg_2, yf97), digits=5)
println("pred_2 = $pred_2\n")

# forecast errors:
e1 = yf97.unem .- pred_1
e2 = yf97.unem .- pred_2

# RMSE and MAE:
rmse1 = sqrt(mean(e1 .^ 2))
println("rmse1 = $rmse1\n")

rmse2 = sqrt(mean(e2 .^ 2))
println("rmse2 = $rmse2\n")

mae1 = mean(abs.(e1))
println("mae1 = $mae1\n")

mae2 = mean(abs.(e2))
println("mae2 = $mae2")

# graph:
plot(yf97.year, yf97.unem, color="black", linewidth=2,
    linestyle=:solid, label="unem", legend=:topleft)
plot!(yf97.year, pred_1, color="black", linewidth=2,
    linestyle=:dash, label="forecast without inflation")
plot!(yf97.year, pred_2, color="black", linewidth=2,
    linestyle=:dashdot, label="forecast with inflation")
ylabel!("unemployment")
xlabel!("time")
savefig("JlGraphs/Example-18-8.pdf")
```

Output of Script 18.5: **Example-18-8.jl**

```
pred_1 = [5.52645, 5.16028, 4.86733, 4.64763, 4.50116, 5.08704, 5.81939]

pred_2 = [5.34847, 4.89645, 4.50914, 4.42518, 4.51606, 4.92354, 5.35027]

rmse1 = 0.5761201545128691

rmse2 = 0.5217548847805118

mae1 = 0.5420143538338797

mae2 = 0.48419578240530825
```

Figure 18.2. Out-of-sample Forecasts for Unemployment



19. Carrying Out an Empirical Project

We are now ready for serious empirical work. Chapter 19 of Wooldridge (2019) discusses the formulation of interesting theories, collection of raw data, and the writing of research papers. We are concerned with the data analysis part of a research project and will cover some aspects of using *Julia* for real research.

This chapter is mainly about a few tips and tricks that might help to make our life easier by organizing the analyses and the output of *Julia* in a systematic way. While we have worked with *Julia* scripts throughout this book, Section 19.1 gives additional hints for using them effectively in larger projects. Section 19.2 shows how the results of our analyses can be written to a text file instead of just being displayed on the screen.

Section 19.3 discusses how Jupyter Notebooks can be used to generate nicely formatted documents that present *Julia* code and output at least in a more structured way, potentially even ready for publication. Therefore we introduce Markdown, a straightforward markup language and $\LaTeX$ a widely used system which was for example used to generate this book. Jupyter Notebooks efficiently use *Julia*, Markdown and $\LaTeX$ together to generate anything between clearly laid out results documentations and complete little research papers that automatically include the analysis results.

19.1. Working with *Julia* Scripts

We already argued in Section 1.1.2 that anything we do in *Julia* or any other statistical package should be done in scripts or the equivalent. In this way, it is always transparent how we generated our results. A typical empirical project has roughly the following steps:

1. Data Preparation: import raw data, recode and generate new variables, create sub-samples, ...
2. Generation of descriptive statistics, distribution of the main variables, ...
3. Estimation of the econometric models
4. Presentation of the results: tables, figures, ...

If we combine all these steps in one *Julia* script, it is very easy for us to understand how we came up with the regression results even a year after we have done the analysis. At least as important: It is also easy for our thesis supervisor, collaborators or journal referees to understand where the results came from and to reproduce them. If we made a mistake at some point or get an updated raw data set, it is easy to repeat the whole analysis to generate new results.

It is crucial to add helpful comments to the *Julia* scripts explaining what is done in each step. Scripts should start with an explanation like the following:

Script 19.1: ultimate-calcs.jl

```
########################################################################
# Project X:
# "The Ultimate Question of Life, the Universe, and Everything"
# Project Collaborators: Mr. H, Mr. B
#
# Julia Script "ultimate-calcs"
# by: F Heiss
# Date of this version: December 1, 2022
########################################################################
# load packages:
using Dates

# create a time stamp:
ts = now()

# print to logfile.txt (write=true resets the logfile before writing output)
# in the provided path (make sure that the folder structure
# you may provide already exists):
open("Jlout/19/logfile.txt", write=true) do io
    println(io, "This is a log file from: \n $ts\n")
end

# the first calculation using the square root function:
result1 = sqrt(1764)
# print to logfile.txt but with keeping the previous results (append=true):
open("Jlout/19/logfile.txt", append=true) do io
    println(io, "result1: $result1\n")
end

# the second calculation reverses the first one:
result2 = result1^2
# print to logfile.txt but with keeping the previous results (append=true):
open("Jlout/19/logfile.txt", append=true) do io
    println(io, "result2: $result2")
end
```

In the next section, we will explain the details of Script 19.1 (`ultimate-calcs.jl`). If a project requires many and/or time-consuming calculations, it might be useful to separate them into several *Julia* scripts. For example, we could have four different scripts corresponding to the steps listed above:

- **`data.jl`**
- **`descriptives.jl`**
- **`estimation.jl`**
- **`results.jl`**

So once the potentially time-consuming data cleaning is done, we don't have to repeat it every time we run regressions. Instead, we save the cleaned data as an intermediary step and load it in subsequent analyses. To avoid confusion, it is highly advisable to document interdependencies. Both **`descriptives.jl`** and **`estimation.jl`** should at the beginning have a comment like:

```
# Depends on data.jl
```

And **`results.jl`** could have a comment like:

```
# Depends on estimation.jl
```

19.2. Logging Output in Text Files

Having the results appear on the screen and being able to copy and paste from there might work for small projects. For larger projects, this is impractical. A straightforward way for writing all results to a file is to use the command **print** (or **println**) and route the output not to the console but a log file. If we want to write the output of a **print** command to a file `logfile.txt`, the basic syntax is:

```
open("logfile.txt", write=true) do io
    print(io, result)
end
```

Script 19.1 (`ultimate-calcs.jl`) gives a demonstration and also explains that the second argument of **open** controls for giving writing access and resetting the log file (**write=true**) or append the results to an existing one (**append=true**). See the documentation for other available options. **end** closes the connection to the log file. We also include a time stamp, to document when we performed our analyses as the following log file resulting from Script 19.1 (`ultimate-calcs.jl`) shows:

File `logfile.txt`

```
This is a log file from:
 2023-03-22T09:47:57.299

result1: 42.0

result2: 1764.0
```

You could also direct all outputs to the log file by only calling **open** once at the beginning. Script 19.2 (`ultimate-calcs2.jl`) demonstrates this alternative and produces the same log file.

Script 19.2: `ultimate-calcs2.jl`

```
# load packages:
using Dates

# create a time stamp:
ts = now()

# print to logfile2.txt (write=true resets the logfile before writing output)
# in the provided path (make sure that the folder structure
# you may provide already exists):
open("Jlout/19/logfile2.txt", write=true) do io
    println(io, "This is a log file from: \n $ts\n")

    # the first calculation using the square root function:
    result1 = sqrt(1764)
    # print to logfile2.txt:
    println(io, "result1: $result1\n")

    # the second calculation reverses the first one:
    result2 = result1^2
    # print to logfile2.txt:
    println(io, "result2: $result2")
end
```

Figure 19.1. Creating a Jupyter Notebook

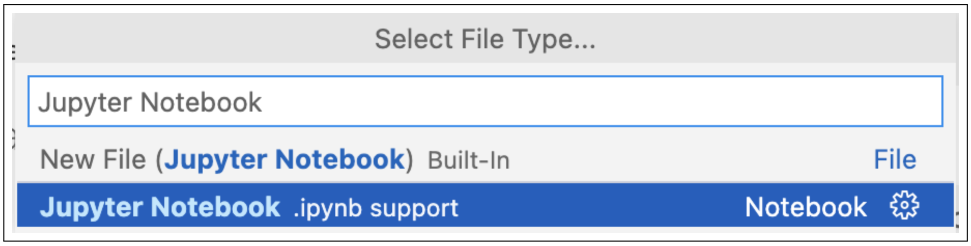
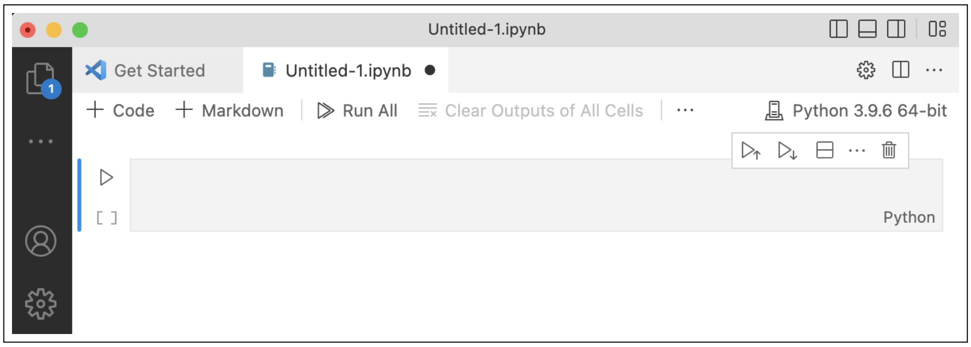


Figure 19.2. An Empty Jupyter Notebook



19.3. Formatted Documents with Jupyter Notebook

Jupyter Notebook is an open source and web based environment that is maintained by the Project Jupyter.[1] A Jupyter Notebook is used to produce documents containing code, formatted text including equations and graphs. You can choose among many formats to export a Jupyter Notebook. Note that although we will use it for *Julia* code only, many other languages like *R* or *Python* are supported.[2]

Visual Studio Code already comes with everything we need to create a Jupyter Notebook. You can also install it manually as explained on `https://jupyter.org/`. In the following, we introduce the interface of Jupyter Notebook and the two important building blocks: `Code` and `Markdown` cells.

19.3.1. Getting Started

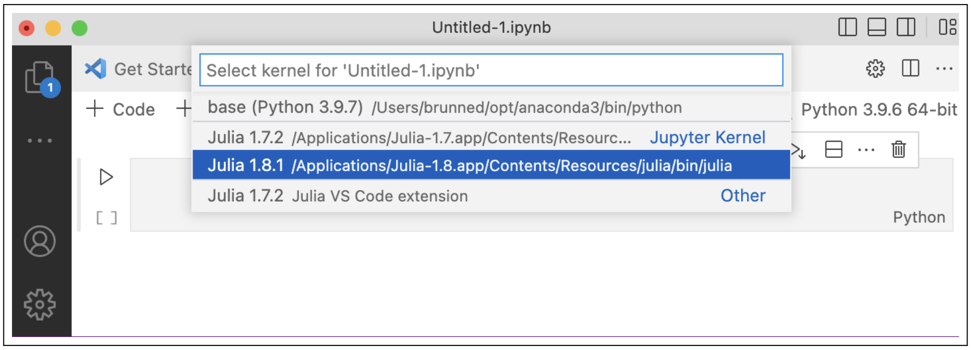
To create a new Jupyter Notebook in Visual Studio Code, use `File`→`New File`, type `Jupyter Notebook` and click on the `.ipynb` entry (also see Figure 19.1).[3] You may be asked to install additional software by Visual Studio Code, if it is your first Jupyter Notebook. This creates an empty Notebook similar as in Figure 19.2. To work with *Julia*, we have to set the right kernel, which can easily be done by clicking on Python 3.9.6 64-bit in the top right in Figure 19.2. The resulting list of languages is shown in Figure 19.3, where *Julia* must be selected to continue.

[1] For more information, see Kluyver, Ragan-Kelley, Pérez, Granger, Bussonnier, Frederic, Kelley, Hamrick, Grout, Corlay, Ivanov, Avila, Abdalla, Willing, and development team (2016).

[2] Actually, the name Jupyter is based on the three languages **Ju**lia, **Pyt**hon and **R**.

[3] Visit `https://code.visualstudio.com/docs/datascience/jupyter-notebooks` for a more detailed introduction.

Figure 19.3. Select *Julia* in an Empty Jupyter Notebook



19.3.2. Cells

Let's start to enter some *Julia* code into the displayed box in Figure 19.3. This box is referred to as a "cell" in a Jupyter Notebook and we choose `3^2` as an exemplary input for such a cell in the upper screenshot in Figure 19.4. You can execute the code by clicking on ▷ to the left of the box and immediately inspect the output in the appearing lines below the cell box (also shown in Figure 19.4). By default, Jupyter Notebook expects you to enter *Julia* code in a cell, which is also visualized by the field in the bottom right saying "`Julia`". You can add more code cells by clicking on + Code.

In the next step we create another cell by clicking on + Markdown. We can now enter text and use Markdown commands to format it. The lower two screenshots of Figure 19.4 give an example. Here we use `**some text**` to print **bold text** and `*` to create a list with bullet points. More useful Markdown commands are explained in the next subsection. After entering the Markdown text click on ✓ in the top left of the box to apply your formatting commands. Instead of printing an output, the cell you previously worked on is replaced by the formatted text. To edit the cell later, just double click on it.

19.3.3. Markdown Basics

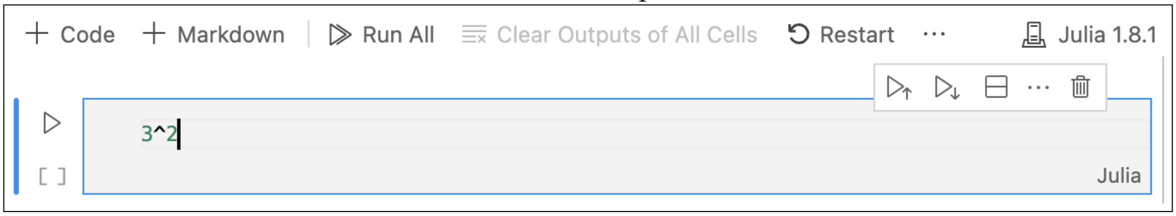
Markdown cells include normal text, formatting instructions and LaTeX equations.[4] There are countless possibilities to create appealing Markdown cells. We can only give a few examples for the most important formatting instructions:

- **`# Header 1`**, **`## Header 2`**, and **`### Header 3`** produce different levels of headers.
- **`*word*`** prints the word in *italics*.
- **`**word**`** prints the word in **bold**.
- **`` ``word`` ``** prints the word in code-like `typewriter font` (obviously **not** for *Julia* code you want to execute).
- We can create lists with bullets using **`*`** at the beginning of a line followed by a whitespace.
- If you are familiar with LaTeX, displayed and inline formulas can be inserted using **`$...$`** and **`$$...$$`** and the usual LaTeX syntax, respectively.

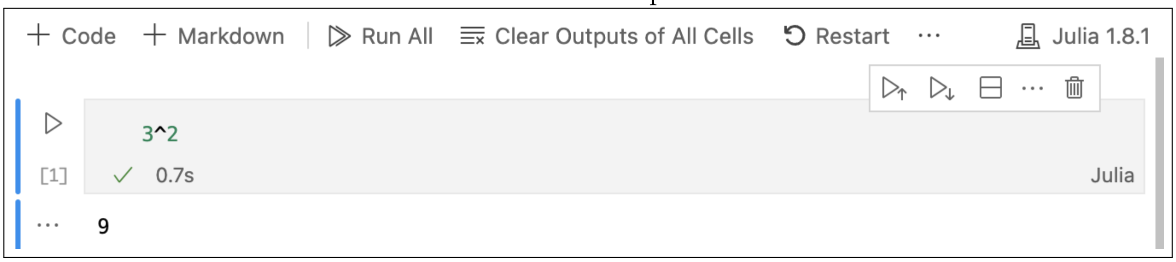
[4] LaTeX is a powerful and free system for generating documents. In economics and other fields with a lot of math involved, it is widely used – in many areas, it is the *de facto* standard. It is also popular for typesetting articles and books. This book is an example for a complex document created by LaTeX. At least basic knowledge of LaTeX is needed to follow the equation related parts.

Figure 19.4. Cells in Jupyter Notebook

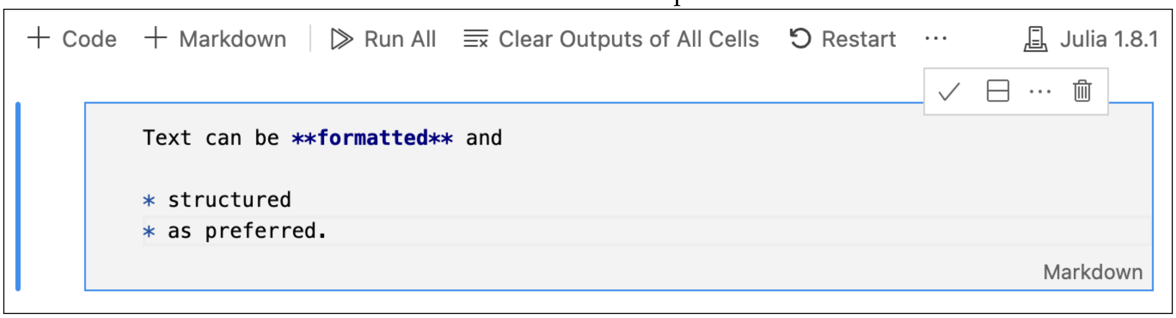
Code Cell Input:



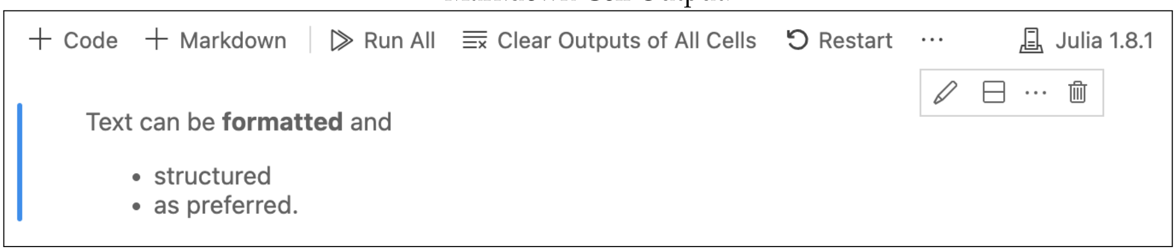
Code Cell Output:



Markdown Cell Input:



Markdown Cell Output:



Different formatting options are demonstrated in the following Jupyter Notebook. It can be downloaded in the `.ipynb` format from `http://www.UPfIE.net`. We start by showing you a collection of all `Code` and `Markdown` cells we entered in our Jupyter Notebook:

File markdown-cell-1.txt
```
# Working with Jupyter Notebook
The following example is based on Script ``Descr-Figures`` from Chapter 1 and
demonstrates the use of **Jupyter Notebooks** to document your work step by step.
We will describe the two most important building blocks:

* basic Markdown commands to format your text in ``Markdown`` cells
* how to import and run Julia code in ``Code`` cells

## Import and Prepare Data
Let's start by loading all packages:
```

File code-cell-1.txt
```
using WooldridgeDatasets, Statistics, DataFrames, FreqTables, Plots
```

File markdown-cell-2.txt
```
In the next step, we import our data and define important variables:
```

File code-cell-2.txt
```
affairs = DataFrame(wooldridge("affairs"))
counts = freqtable(affairs.kids)
labels = ["no", "yes"]

print(counts)
```

File markdown-cell-3.txt
```
## Analyse Data
### View your Data
To get an overview you could use ``first(affairs, 5)``.

### Calculate Descriptive Statistics
Now we are interested in printing out the average age.
We start with its definition and use LaTeX to enter
the equation:
$$ \bar{x} = \frac{1}{N} \sum_{i=1}^N x_{i} $$
The resulting Julia code gives:
```

File code-cell-3.txt
```
age_mean = mean(affairs.age)
print(age_mean)
```

File markdown-cell-4.txt
```
### Produce Graphic Results
In Chapter 1, we saw how to produce a pie chart. Let's repeat it here:
```

File code-cell-4.txt
```
pie(labels, counts)
```

File markdown-cell-5.txt
````
You can also show Julia code without executing it. You can use ``inline code``,
or for longer paragraphs
```julia
bar(labels, counts)
```
````

We exported the Jupyter Notebook into `PDF` and produced the following document:

Figure 19.5. Example of an Exported Jupyter Notebook

Working with Jupyter Notebook

The following example is based on Script `Descr-Figures` from Chapter 1 and demonstrates the use of **Jupyter Notebooks** to document your work step by step. We will describe the two most important building blocks:

- basic Markdown commands to format your text in `Markdown` cells
- how to import and run Julia code in `Code` cells

Import and Prepare Data

Let's start by loading all packages:

In [ ]:
```
using WooldridgeDatasets, Statistics, DataFrames, FreqTables, Plots
```

In the next step, we import our data and define important variables:

In [ ]:
```
affairs = DataFrame(wooldridge("affairs"))
counts = freqtable(affairs.kids)
labels = ["no", "yes"]

print(counts)
```

```
[171, 430]
```

Analyse Data

View your Data

To get an overview you could use `first(affairs, 5)` .

Calculate Descriptive Statistics

Now we are interested in printing out the average age. We start with its definition and use LaTeX to enter the equation:

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

The resulting Julia code gives:

In [ ]:
```
age_mean = mean(affairs.age)
print(age_mean)
```

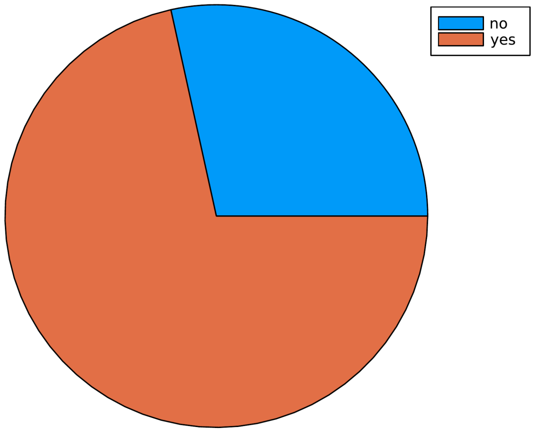
```
32.48752079866888
```

Figure 19.6. Example of an Exported Jupyter Notebook (cont'ed)

Produce Graphic Results

In Chapter 1, we saw how to produce a pie chart. Let's repeat it here:

In [ ]:
```
pie(labels, counts)
```



You can also show Julia code without executing it. You can use `inline code`, or for longer paragraphs

```
bar(labels, counts)
```

Part IV.

Appendices

Julia Scripts

1. Scripts Used in Chapter 01

Script 1.1: `First-Julia-Script.jl`

```
# This is a comment.
# in the next line, we try to enter Shakespeare:
"To be, or not to be: that is the question"
# let's try some sensible math:
sqrt(16)
print((1 + 2) * 5)
```

Script 1.2: `Julia-as-a-Calculator.jl`

```
result1 = 1 + 1
print("result1 = $result1\n")

result2 = 5 * (4 - 1)^2
println("result2 = $result2")

result3 = [result1, result2]
print("result3:\n $result3")
```

Script 1.3: `Package-Statistics.jl`

```
using Statistics

a = [2, 6, 4, 9, 1]

a_mean = mean(a)
println("a_mean = $a_mean")

a_var = Statistics.var(a)
println("a_var = $a_var")
```

Script 1.4: `Objects-in-Julia.jl`

```
result1 = 1 + 1
# determine the type:
type_result1 = typeof(result1)
# print the result:
println("type_result1: $type_result1\n")

result2 = 2.5
type_result2 = typeof(result2)
println("type_result2: $type_result2\n")

result3 = "To be, or not to be: that is the question"
type_result3 = typeof(result3)
println("type_result3: $type_result3")
```

Script 1.5: `Arrays.jl`

```
# define arrays:
testarray1D = [1, 5, 41.3, 2.0]
println("type(testarray1D): $(typeof(testarray1D))\n")

testarray2D = [4 9 8 3
    2 6 3 2
    1 1 7 4]

# same as:
#testarray2D = [4 9 8 3; 2 6 3 2; 1 1 7 4]
#testarray2D = [[4 9 8 3]
#    [ 2 6 3 2]
#    [ 1 1 7 4]]
#testarray2D = [[4, 2, 1] [9, 6, 1] [8, 3, 7] [3, 2, 4]]

# get dimensions of testarray2D:
dim = size(testarray2D)
println("dim: $dim\n")

# access elements by indices:
third_elem = testarray1D[3]
println("third_elem = $third_elem\n")

second_third_elem = testarray2D[2, 3] # element in 2nd row and 3rd column
println("second_third_elem = $second_third_elem\n")

second_to_third_col = testarray2D[:, 2:3] # each row in the 2nd and 3rd column
println("second_to_third_col = $second_to_third_col\n")

last_col = testarray2D[:, end] # each row in the last column
println("last_col = $last_col\n")

# access elements by array:
first_third_elem = testarray1D[[1, 3]]
println("first_third_elem: $first_third_elem\n")

# same with Boolean array:
first_third_elem2 = testarray1D[[true, false, true, false]]
println("first_third_elem2 = $first_third_elem2\n")

k = [[true false false false]
    [false false true false]
    [true false true false]]
elem_by_index = testarray2D[k] # 1st elem in 1st row, 1st elem in 3rd row...
print("elem_by_index: $elem_by_index")
```

Script 1.6: `Array-Copy.jl`

```
# define arrays:
testarray = [1, 5, 41.3, 2.0]

# be careful with changes on variables pointing on testarray:
duplicate_array = testarray
duplicate_array[3] = 10000
println("duplicate_array: $duplicate_array\n")
println("testarray: $testarray\n")

# work on a copy of example_list:
testarray = [1, 5, 41.3, 2.0]
```

```
duplicate_array = deepcopy(testarray)
duplicate_array[3] = 10000
println("duplicate_array: $duplicate_array\n")
println("testarray: $testarray")
```

Script 1.7: `Array-Functions.jl`

```
# define arrays:
vec1 = [1, 4, 64, 36]
mat1 = [4 9 8 3
    2 6 3 2
    1 1 7 4]

# apply some functions:
sort!(vec1)
println("vec1: $vec1\n")

vec2 = sqrt.(vec1)
vec3 = vec1 .+ vec2
println("vec3: $vec3\n")

# get dimensions of mat1:
dim_mat1 = size(mat1)
println("dim_mat1: $dim_mat1")
```

Script 1.8: `Array-SpecialCases.jl`

```
# initialize matrix with each element set to zero:
zero_mat = zeros(4, 3)
println("zero_mat: \n$zero_mat\n")

# initialize matrix with each element set to one:
one_mat = ones(2, 5)
println("one_mat: \n$one_mat\n")

# uninitialized matrix (filled with arbitrary nonsense elements):
empty_mat = Array{Float64}(undef, 2, 2)
println("empty_mat: \n$empty_mat")
```

Script 1.9: `Dicts.jl`

```
# define and print a dict:
var1 = ["Florian", "Daniel"]
var2 = [96, 49]
example_dict = Dict("name" => var1, "points" => var2)
println("example_dict: \n$example_dict\n")

# get data type:
type_example_dict = typeof(example_dict)
println("type_example_dict: $type_example_dict\n")

# access "points":
points_all = example_dict["points"]
println("points_all: $points_all\n")

# access "points" of Daniel:
points_daniel = example_dict["points"][2]
println("points_daniel: $points_daniel\n")

# add 4 to "points" of Daniel:
```

```
example_dict["points"][2] = example_dict["points"][2] + 4
println("example_dict: \n$example_dict\n")

# add a new component "grade":
example_dict["grade"] = [1.3, 4.0]

# delete component "points":
delete!(example_dict, "points")
print("example_dict: \n$example_dict\n")
```

Script 1.10: `Matrix-Operations.jl`

```
# define matrices:
mat1 = [4 9 8
        2 6 3]
mat2 = [1 5 2
        6 6 0
        4 8 3]

# use exp() and apply it to each element:
result1 = exp.(mat1)
result1_rounded = round.(result1, digits=4)
println("result1_rounded: \n$result1_rounded\n")

result2 = mat1 .+ mat2[1:2, :]
println("result2: $result2\n")

# use another function:
mat1_tr = transpose(mat1) #or simply: mat1'
println("mat1_tr: $mat1_tr\n")

# matrix algebra:
matprod = mat1 * mat2
println("matprod: $matprod")
```

Script 1.11: `DataFrames.jl`

```
using DataFrames

# define a DataFrame:
icecream_sales = [30, 40, 35, 130, 120, 60]
weather_coded = [0, 1, 0, 1, 1, 0]
customers = [2000, 2100, 1500, 8000, 7200, 2000]
df = DataFrame(
    icecream_sales=icecream_sales,
    weather_coded=weather_coded,
    customers=customers
)

# print the DataFrame
println("df: \n$df\n")

# access columns by variable reference:
subset1 = df[!, [:icecream_sales, :customers]]
println("subset1: \n$subset1\n")

# access second to fourth row:
subset2 = df[2:4, :]
println("subset2: \n$subset2\n")
```

```
# access rows and columns by variable integer positions:
subset3 = df[2:4, 1:2]
println("subset3: \n$subset3\n")

# access rows by variable integer positions:
subset4 = df[2:4, [:icecream_sales, :weather_coded]]
println("subset4: \n$subset4")
```

Script 1.12: `DataFrames-Functions.jl`

```
using DataFrames, CategoricalArrays, Statistics

# define a DataFrame:
icecream_sales = [30, 40, 35, 130, 120, 60]
weather_coded = [0, 1, 0, 1, 1, 0]
customers = [2000, 2100, 1500, 8000, 7200, 2000]
df = DataFrame(
    icecream_sales=icecream_sales,
    weather_coded=weather_coded,
    customers=customers
)

# get some descriptive statistics:
descr_stats = describe(df)
println("descr_stats: \n$descr_stats\n")

# add one observation at the end in-place:
push!(df, [50, 1, 3000])
println("df: \n$df\n")

# extract observations with more than 2500 customers:
subset_df = subset(df, :customers => ByRow(>(2500)))
println("subset_df: \n$subset_df\n")

# use a CategoricalArray object to attach labels (0 = bad; 1 = good):
df.weather = recode(df[!, :weather_coded], 0 => "bad", 1 => "good")
println("df \n$df\n")

# mean sales for each weather category by
# grouping and splitting data:
grouped_data = groupby(df, :weather)
# apply the mean to icecream_sales and combine the results:
group_means = combine(grouped_data, :icecream_sales => mean)
println("group_means: \n$group_means")
```

Script 1.13: `PyCall-Simple.jl`

```
using PyCall

# define a block of Python Code:
py"""
import numpy as np

# define arrays in numpy:
mat1 = np.array([[4, 9, 8],
                 [2, 6, 3]])
mat2 = np.array([[1, 5, 2],
                 [6, 6, 0],
                 [4, 8, 3]])
# matrix algebra:
```

```
matprod_py = mat1 @ mat2
"""

# automatic type conversion from Python to Julia:
matprod = py"matprod_py"
matprod_type = typeof(matprod)
println("matprod_type: $matprod_type\n")
println("matprod: $matprod")
```

Script 1.14: `PyCall-Alternative.jl`

```
using PyCall

# using pyimport to work with modules:
np = pyimport("numpy")

# define matrices in Julia:
mat1 = [4 9 8
        2 6 3]
mat2 = [1 5 2
        6 6 0
        4 8 3]

# ... and pass them to numpys dot function:
matprod = np.dot(mat1, mat2)
println("matprod: $matprod\n")

matprod_type = typeof(matprod)
println("matprod_type: $matprod_type")
```

Script 1.15: `Wooldridge.jl`

```
using WooldridgeDatasets, DataFrames

# load data:
wage1 = DataFrame(wooldridge("wage1"))

# get type:
type_wage1 = typeof(wage1)
println("type_wage1: $type_wage1\n")

# get first four observations and first eight variables:
preview_wage1 = wage1[1:4, 1:8]
println("preview_wage1: \n$preview_wage1")
```

Script 1.16: `Import-Export.jl`

```
using DataFrames, CSV

# import a .CSV file with CSV.read:
df1 = CSV.read("data/sales.csv", DataFrame, delim=",",
    header=["year", "product1", "product2", "product3"])
println("df1: \n$df1\n")

# import a .txt file with CSV.read:
df2 = CSV.read("data/sales.txt", DataFrame, delim=" ")
println("df2: \n$df2\n")

# add a row to df1:
push!(df1, [2014, 10, 8, 2])
```

```
println("df1: \n$df1")

# export with CSV.write:
CSV.write("data/sales2.csv", df1)
```

Script 1.17: `Import-StockData.jl`

```
using DataFrames, Dates, MarketData

# download data for "F" (= Ford) and define start and end:
ticker = "F"
start_date = DateTime(2007, 12, 31)
end_date = DateTime(2017, 01, 01)

# import data as DataFrame:
F_data = DataFrame(yahoo(ticker,
    YahooOpt(period1=start_date, period2=end_date)))

preview_F_data = first(F_data, 5)
println("preview_F_data: \n$preview_F_data")
```

Script 1.18: `Graphs-Basics.jl`

```
using Plots

# create data:
x = [1, 3, 4, 7, 8, 9]
y = [0, 3, 6, 9, 7, 8]

# plot and save:
plot(x, y, color=:black)
savefig("JlGraphs/Graphs-Basics-a.pdf")

# scatter and save:
scatter(x, y, color=:black, markershape=:dtriangle, legend=false)
savefig("JlGraphs/Graphs-Basics-b.pdf")
```

Script 1.19: `Graphs-Basics2.jl`

```
using Plots

# create data:
x = [1, 3, 4, 7, 8, 9]
y = [0, 3, 6, 9, 7, 8]

# plot and save:
plot(x, y, color=:black, linestyle=:dash, legend=false)
savefig("JlGraphs/Graphs-Basics-c.pdf")

plot(x, y, color=:black, linestyle=:dot, legend=false)
savefig("JlGraphs/Graphs-Basics-d.pdf")

plot(x, y, color=:black, linestyle=:solid, linewidth=3, legend=false)
savefig("JlGraphs/Graphs-Basics-e.pdf")

plot(x, y, color=:black, markershape=:circle, legend=false)
savefig("JlGraphs/Graphs-Basics-f.pdf")
```

Script 1.20: `Graphs-Functions.jl`

```
using Plots, Distributions

# support of quadratic function
# (creates an array with 100 equispaced elements from -3 to 2):
x1 = range(start=-3, stop=2, length=100)
# function values for all these values:
y1 = x1 .^ 2

# plot quadratic function:
plot(x1, y1, linestyle=:solid, color=:black, legend=false)
savefig("JlGraphs/Graphs-Functions-a.pdf")

# same for normal density:
x2 = range(-4, 4, length=100)
y2 = pdf.(Normal(), x2)

# plot normal density:
plot(x2, y2, linestyle=:solid, color=:black, legend=false)
savefig("JlGraphs/Graphs-Functions-b.pdf")
```

Script 1.21: `Graphs-BuildingBlocks.jl`

```
using Plots, Distributions

# support for all normal densities:
x = range(-4, 4, length=100)
# get different density evaluations:
y1 = pdf.(Normal(), x)
y2 = pdf.(Normal(1, 0.5), x)
y3 = pdf.(Normal(0, 2), x)

# plot:
plot(x, y1, linestyle=:solid, color=:black, label="standard normal")
plot!(x, y2, linestyle=:dash, color=:black,
    linealpha=0.6, label="mu = 1, sigma = 0.5")
plot!(x, y3, linestyle=:dot, color=:black,
    linealpha=0.3, label="mu = 0, sigma = 2")
xlims!(-3, 4)
title!("Normal Densities")
ylabel!("phi(x)")
xlabel!("x")
savefig("JlGraphs/Graphs-BuildingBlocks.pdf")
```

Script 1.22: `Graphs-Export.jl`

```
using Plots, Distributions

# support for all normal densities:
x = range(-4, 4, length=100)
# get different density evaluations:
y1 = pdf.(Normal(), x)
y2 = pdf.(Normal(0, 3), x)

# plot (a):
plot(legend=false, size=(400, 600))
plot!(x, y1, linestyle=:solid, color=:black)
plot!(x, y2, linestyle=:dash, color=:black, linealpha=0.3)
savefig("JlGraphs/Graphs-Export-a.pdf")
```

```
# plot (b):
plot(legend=false, size=(600, 400))
plot!(x, y1, linestyle=:solid, color=:black)
plot!(x, y2, linestyle=:dash, color=:black, linealpha=0.3)
savefig("JlGraphs/Graphs-Export-b.png")
```

Script 1.23: `Descr-Tables.jl`

```
using WooldridgeDatasets, DataFrames, CategoricalArrays, FreqTables

affairs = DataFrame(wooldridge("affairs"))

# attach labels to kids and convert it to a categorical variable:
affairs.haskids = categorical(
    recode(affairs.kids, 0 => "no", 1 => "yes")
)

# ... and ratemarr (for example: 1 = "very unhappy", 2 = "unhappy",...):
affairs.marriage = categorical(
    recode(affairs.ratemarr,
        1 => "very unhappy",
        2 => "unhappy",
        3 => "average",
        4 => "happy",
        5 => "very happy"
    )
)

# frequency table (alphabetical order of elements):
ft_marriage = freqtable(affairs.marriage)
println("ft_marriage: \n$ft_marriage\n")

# frequency table with groupby:
ft_groupby = combine(
    groupby(affairs, :haskids),
    nrow)
println("ft_groupby: \n$ft_groupby\n")

# contingency tables with absolute and relative values:
ct_all_abs = freqtable(affairs.marriage, affairs.haskids)
println("ct_all_abs: \n$ct_all_abs\n")

ct_all_rel = proptable(affairs.marriage, affairs.haskids)
println("ct_all_rel: \n$ct_all_rel\n")

# share within "marriage" (i.e. within a row):
ct_row = proptable(affairs.marriage, affairs.haskids, margins=1)
println("ct_row: \n$ct_row\n")

# share within "haskids" (i.e. within a column):
ct_col = proptable(affairs.marriage, affairs.haskids, margins=2)
println("ct_col: \n$ct_col")
```

Script 1.24: `Descr-Figures.jl`

```
using WooldridgeDatasets, DataFrames, Plots, StatsPlots,
    FreqTables, CategoricalArrays

affairs = DataFrame(wooldridge("affairs"))
```

```
# attach labels to kids and convert it to a categorical variable:
affairs.haskids = categorical(
    recode(affairs.kids, 0 => "no", 1 => "yes")
)

# ... and ratemarr (for example: 1 = "very unhappy", 2 = "unhappy",...):
affairs.marriage = categorical(
    recode(affairs.ratemarr,
        1 => "very unhappy",
        2 => "unhappy",
        3 => "average",
        4 => "happy",
        5 => "very happy"
    )
)

# counts for all graphs:
counts_m = sort(freqtable(affairs.marriage), rev=true)
levels_counts_m = String.(collect(keys(counts_m.dicts[1])))
colors_m = [:grey60, :grey50, :grey40, :grey30, :grey20]

ct_all_abs = freqtable(affairs.marriage, affairs.haskids)
levels_counts_all = String.(collect(keys(ct_all_abs.dicts[1])))
colors_all = [:grey80 :grey50]

# pie chart (a):
pie(levels_counts_m, counts_m, color=colors_m)
savefig("JlGraphs/Descr-Pie.pdf")

# bar chart (b):
bar(levels_counts_m, counts_m, color=:grey, legend=false)
savefig("JlGraphs/Descr-Bar1.pdf")

# stacked bar plot (c):
groupedbar(ct_all_abs, bar_position=:stack,
    color=colors_all, label=["no" "yes"])
xticks!(1:5, levels_counts_all)
savefig("JlGraphs/Descr-Bar2.pdf")

# grouped bar plot (d):
groupedbar(ct_all_abs, bar_position=:dodge,
    color=colors_all, label=["no" "yes"])
xticks!(1:5, levels_counts_all)
savefig("JlGraphs/Descr-Bar3.pdf")
```

Script 1.25: `Histogram.jl`

```
using WooldridgeDatasets, Plots, DataFrames

ceosal1 = DataFrame(wooldridge("ceosal1"))

# extract roe:
roe = ceosal1.roe

# histogram with counts (a):
histogram(roe, color=:grey, legend=false)
ylabel!("Counts")
xlabel!("roe")
savefig("JlGraphs/Histogram1.pdf")
```

```
# histogram with density and explicit breaks (b):
breaks = [0, 5, 10, 20, 30, 60]
histogram(roe, color=:grey,
    bins=breaks,
    normalize=true,
    legend=false)
xlabel!("roe")
ylabel!("Density")
savefig("JlGraphs/Histogram2.pdf")
```

Script 1.26: KDensity.jl

```
using WooldridgeDatasets, DataFrames, Plots, KernelDensity

ceosal1 = DataFrame(wooldridge("ceosal1"))

# extract roe:
roe = ceosal1.roe

# estimate kernel density:
kde_est = KernelDensity.kde(roe)

# kernel density (a):
plot(kde_est.x, kde_est.density, color=:black, linewidth=2, legend=false)
ylabel!("density")
xlabel!("roe")
savefig("JlGraphs/Density1.pdf")

# kernel density with overlayed histogram (b):
histogram(roe, color="grey", normalize=true, legend=false)
plot!(kde_est.x, kde_est.density, color=:black, linewidth=2)
ylabel!("density")
xlabel!("roe")
savefig("JlGraphs/Density2.pdf")
```

Script 1.27: Descr-ECDF.jl

```
using WooldridgeDatasets, DataFrames, Plots

ceosal1 = DataFrame(wooldridge("ceosal1"))

# extract roe:
roe = ceosal1.roe

# calculate ECDF:
x = sort(roe)
n = length(x)
y = range(start=1, stop=n) / n

# plot a step function:
plot(x, y, linetype=:steppre, color=:black, legend=false)
xlabel!("roe")
savefig("JlGraphs/ecdf.pdf")
```

Script 1.28: Descr-Stats.jl

```
using WooldridgeDatasets, DataFrames, Statistics

ceosal1 = DataFrame(wooldridge("ceosal1"))
```

```
# extract roe and salary:
roe = ceosal1.roe
salary = ceosal1.salary

# sample average:
roe_mean = mean(roe)
println("roe_mean = $roe_mean\n")

# sample median:
roe_med = median(roe)
println("roe_med = $roe_med\n")

# corrected standard deviation (n-1 scaling):
roe_std = std(roe)
println("roe_st = $roe_std\n")

# correlation with roe:
roe_corr = cor(roe, salary)
println("roe_corr = $roe_corr\n")

# correlation matrix with roe:
roe_corr_mat = cor(hcat(roe, salary))
println("roe_corr_mat: \n$roe_corr_mat")
```

Script 1.29: `Descr-Boxplot.jl`

```
using WooldridgeDatasets, DataFrames, StatsPlots

ceosal1 = DataFrame(wooldridge("ceosal1"))
# extract roe and salary:
roe = ceosal1.roe
consprod = ceosal1.consprod

# plotting descriptive statistics:
boxplot(roe, orientation=:h,
    linecolor=:black, color=:white, legend=false)
yticks!([1], [""])
ylabel!("roe")
savefig("JlGraphs/Boxplot1.pdf")

# plotting descriptive statistics (logical indexing):
roe_cp0 = roe[consprod.==0]
roe_cp1 = roe[consprod.==1]
boxplot([roe_cp0, roe_cp1], linecolor=:black,
    color=:white, legend=false)
xticks!([1, 2], ["consprod=0", "consprod=1"])
ylabel!("roe")
savefig("JlGraphs/Boxplot2.pdf")
```

Script 1.30: `PMF-binom.jl`

```
using Distributions

# pedestrian approach:
p1 = binomial(10, 2) * (0.2^2) * (0.8^8)
println("p1 = $p1\n")

# package function:
p2 = pdf(Binomial(10, 0.2), 2)
println("p2 = $p2")
```

Script 1.31: `PMF-example.jl`

```
using Distributions, DataFrames, Plots

# PMF for all values between 0 and 10:
x = 0:10
fx = pdf.(Binomial(10, 0.2), x)

# collect values in DataFrame:
result = DataFrame(x=x, fx=fx)
println("result: \n$result")

# plot:
bar(x, fx, color=:grey, alpha=0.6, legend=false)
xlabel!("x")
ylabel!("fx")
savefig("JlGraphs/PMF-example.pdf")
```

Script 1.32: `PDF-example.jl`

```
using Plots, Distributions

# support of normal density:
x_range = range(-4, 4, length=100)

# PDF for all these values:
pdf_normal = pdf.(Normal(), x_range)

# plot:
plot(x_range, pdf_normal, color=:black, legend=false)
xlabel!("x")
ylabel!("dx")
savefig("JlGraphs/PDF-example.pdf")
```

Script 1.33: `CDF-example.jl`

```
using Distributions

# binomial CDF:
p1 = cdf(Binomial(10, 0.2), 3)
println("p1 = $p1\n")

# normal CDF:
p2 = cdf(Normal(), 1.96) - cdf(Normal(), -1.96)
println("p2 = $p2")
```

Script 1.34: `Example-B-6.jl`

```
using Distributions

# first example using the transformation:
p1_1 = cdf(Normal(), 2 / 3) - cdf(Normal(), -2 / 3)
println("p1_1 = $p1_1\n")

# first example working directly with the distribution of X:
p1_2 = cdf(Normal(4, 3), 6) - cdf(Normal(4, 3), 2)
println("p1_2 = $p1_2\n")

# second example:
p2 = 1 - cdf(Normal(4, 3), 2) + cdf(Normal(4, 3), -2)
println("p2 = $p2")
```

Script 1.35: `CDF-figure.jl`

```
using Distributions, Plots

# binomial CDF:
x_binom = range(-1, 10, length=100)
cdf_binom = cdf.(Binomial(10, 0.2), x_binom)

plot(x_binom, cdf_binom, linetype=:steppre, color=:black, legend=false)
xlabel!("x")
ylabel!("Fx")
savefig("JlGraphs/CDF-figure-discrete.pdf")

# normal CDF:
x_norm = range(-4, 4, length=1000)
cdf_norm = cdf.(Normal(), x_norm)

plot(x_norm, cdf_norm, color=:black, legend=false)
xlabel!("x")
ylabel!("Fx")
savefig("JlGraphs/CDF-figure-cont.pdf")
```

Script 1.36: `Quantile-example.jl`

```
using Distributions

q_975 = quantile(Normal(), 0.975)
println("q_975 = $q_975")
```

Script 1.37: `Sample-Bernoulli.jl`

```
using Distributions

sample = rand(Bernoulli(0.5), 10)
println("sample: $sample")
```

Script 1.38: `Sample-Norm.jl`

```
using Distributions

sample = rand(Normal(), 6)
sample_rounded = round.(sample, digits=5)
println("sample_rounded: $sample_rounded")
```

Script 1.39: `Random-Numbers.jl`

```
using Distributions, Random

Random.seed!(12345)
# sample from a standard normal RV with sample size n=3:
sample1 = rand(Normal(), 3)
println("sample1: $sample1\n")

# a different sample from the same distribution:
sample2 = rand(Normal(), 3)
println("sample2: $sample2\n")

# set the seed of the random number generator and take two samples:
Random.seed!(54321)
sample3 = rand(Normal(), 3)
println("sample3: $sample3\n")
```

```
sample4 = rand(Normal(), 3)
println("sample4: $sample4\n")

# reset the seed to the same value to get the same samples again:
Random.seed!(54321)
sample5 = rand(Normal(), 3)
println("sample5: $sample5\n")

sample6 = rand(Normal(), 3)
println("sample6: $sample6")
```

Script 1.40: `Example-C-2.jl`

```
using Distributions

# manually enter raw data from Wooldridge, Table C.3:
SR87 = [10, 1, 6, 0.45, 1.25, 1.3, 1.06, 3, 8.18, 1.67,
    0.98, 1, 0.45, 5.03, 8, 9, 18, 0.28, 7, 3.97]
SR88 = [3, 1, 5, 0.5, 1.54, 1.5, 0.8, 2, 0.67, 1.17, 0.51,
    0.5, 0.61, 6.7, 4, 7, 19, 0.2, 5, 3.83]

# calculate change:
Change = SR88 .- SR87

# ingredients to CI formula:
avgCh = mean(Change)
println("avgCh = $avgCh\n")

n = length(Change)
sdCh = std(Change)
se = sdCh / sqrt(n)
println("se = $se\n")

c = quantile(TDist(n - 1), 0.975)
println("c = $c\n")

# confidence interval:
lowerCI = avgCh - c * se
println("lowerCI = $lowerCI\n")
upperCI = avgCh + c * se
println("upperCI = $upperCI")
```

Script 1.41: `Example-C-3.jl`

```
using WooldridgeDatasets, DataFrames, Distributions

audit = DataFrame(wooldridge("audit"))
y = audit.y

# ingredients to CI formula:
avgy = mean(y)
n = length(y)
sdy = std(y)
se = sdy / sqrt(n)
c95 = quantile(Normal(), 0.975)
c99 = quantile(Normal(), 0.995)

# 95% confidence interval:
lowerCI95 = avgy - c95 * se
println("lowerCI95 = $lowerCI95\n")
```

```
upperCI95 = avgy + c95 * se
println("upperCI95 = $upperCI95\n")

# 99% confidence interval:
lowerCI99 = avgy - c99 * se
println("lowerCI99 = $lowerCI99\n")

upperCI99 = avgy + c99 * se
println("upperCI99 = $upperCI99")
```

Script 1.42: `Critical-Values-t.jl`

```
using Distributions, DataFrames

# degrees of freedom = n-1:
df = 19

# significance levels:
alpha_one_tailed = [0.1, 0.05, 0.025, 0.01, 0.005, 0.001]
alpha_two_tailed = alpha_one_tailed * 2

# critical values & table:
CV = quantile.(TDist(df), 1 .- alpha_one_tailed)
table = DataFrame(alpha_one_tailed=alpha_one_tailed,
    alpha_two_tailed=alpha_two_tailed,
    CV=CV)
println("table: \n$table")
```

Script 1.43: `Example-C-5.jl`

```
using WooldridgeDatasets, DataFrames, Distributions, HypothesisTests

audit = DataFrame(wooldridge("audit"))
y = audit.y

# automated calculation of t statistic for H0 (mu=0):
test_auto = OneSampleTTest(y, 0)
t_auto = test_auto.t # access test statistic
p_auto = pvalue(test_auto, tail=:left)  # access one-sided p value
println("t_auto = $t_auto\n")
println("p_auto = $p_auto\n")

# manual calculation of t statistic for H0 (mu=0):
avgy = mean(y)
n = length(y)
sdy = std(y)
se = sdy / sqrt(n)
t_manual = avgy / se
println("t_manual = $t_manual\n")

# critical values for t distribution with n-1=240 d.f.:
alpha_one_tailed = [0.1, 0.05, 0.025, 0.01, 0.005, 0.001]
CV = quantile(TDist(n - 1), 1 .- alpha_one_tailed)
table = DataFrame(alpha_one_tailed=alpha_one_tailed, CV=CV)
println("table: \n$table")
```

Script 1.44: `Example-C-6.jl`

```
using Distributions, HypothesisTests
```

```
# manually enter raw data from Wooldridge, Table C.3:
SR87 = [10, 1, 6, 0.45, 1.25, 1.3, 1.06, 3, 8.18, 1.67,
        0.98, 1, 0.45, 5.03, 8, 9, 18, 0.28, 7, 3.97]
SR88 = [3, 1, 5, 0.5, 1.54, 1.5, 0.8, 2, 0.67, 1.17, 0.51,
        0.5, 0.61, 6.7, 4, 7, 19, 0.2, 5, 3.83]
Change = SR88 .- SR87

# automated calculation of t statistic for H0 (mu=0):
test_auto = OneSampleTTest(Change, 0)
t_auto = test_auto.t
p_auto = pvalue(test_auto, tail=:left)
println("t_auto = $t_auto\n")
println("p_auto = $p_auto\n")

# manual calculation of t statistic for H0 (mu=0):
avgCh = mean(Change)
n = length(Change)
sdCh = std(Change)
se = sdCh / sqrt(n)
t_manual = avgCh / se
println("t_manual = $t_manual\n")

# manual calculation of p value for H0 (mu=0):
p_manual = cdf(TDist(n - 1), t_manual)
println("p_manual = $p_manual")
```

Script 1.45: `Example-C-7.jl`

```
using WooldridgeDatasets, DataFrames, Distributions, HypothesisTests

audit = DataFrame(wooldridge("audit"))
y = audit.y

# automated calculation of t statistic for H0 (mu=0):
test_auto = OneSampleTTest(y, 0)
t_auto = test_auto.t
p_auto = pvalue(test_auto, tail=:left)
println("t_auto = $t_auto\n")
println("p_auto = $p_auto\n")

# manual calculation of t statistic for H0 (mu=0):
avgy = mean(y)
n = length(y)
sdy = std(y)
se = sdy / sqrt(n)
t_manual = avgy / se
println("t_manual = $t_manual\n")

# manual calculation of p value for H0 (mu=0):
p_manual = cdf(TDist(n - 1), t_manual)
println("p_manual = $p_manual")
```

Script 1.46: `Adv-Loops.jl`

```
seq = [1, 2, 3, 4, 5, 6]
for i in seq
    if i < 4
        println(i^3)
    else
        println(i^2)
```

```
    end
end
```

Script 1.47: `Adv-Loops2.jl`

```
seq = [1, 2, 3, 4, 5, 6]

for i in eachindex(seq)
    if seq[i] < 4
        println(seq[i]^3)
    else
        println(seq[i]^2)
    end
end
```

Script 1.48: `Adv-Functions.jl`

```
# define function:
function mysqrt(x)
    if x >= 0
        result = x^0.5
    else
        result = "You fool!"
    end
    return result
end

# call function and save result:
result1 = mysqrt(4)
println("result1 = $result1\n")

result2 = mysqrt(-1.5)
println("result2 = $result2")
```

Script 1.49: `Adv-Functions-MultArg.jl`

```
# define function (only positional arguments):
function mysqrt_pos(x, add)
    if x >= 0
        result = x^0.5 + add
    else
        result = "You fool!"
    end
    return result
end

# define function ("x" as positional and "add" as keyword argument):
function mysqrt_kwd(x; add)
    if x >= 0
        result = x^0.5 + add
    else
        result = "You fool!"
    end
    return result
end

# call functions:
result1 = mysqrt_pos(4, 3)
println("result1 = $result1")
# mysqrt_pos(4, add = 3) is not valid
```

```
result2 = mysqrt_kwd(4, add=3)
println("result2 = $result2")
# mysqrt_kwd(4, 3) is not valid
```

Script 1.50: **Adv-Performance.jl**

```
using Random, Distributions
# set the random seed:
Random.seed!(12345)

function simMean(n, reps)
    ybar = zeros(reps)
    for j in 1:reps
        # sample from normal distribution of size n
        sample = rand(Normal(), n)
        ybar[j] = mean(sample)
    end
    return ybar
end

# call the function and measure time:
n = 100
reps = 10000
stats = @timed simMean(n, reps);
runTime = stats.time
println("runTime = $runTime")
```

Script 1.51: **Adv-Performance-Jl-Figure.jl**

```
using Random, DataFrames, Distributions, Plots, BenchmarkTools, CSV
# set the random seed:
Random.seed!(12345)

function simMean(n, reps)
    ybar = zeros(reps)
    for j in 1:reps
        # sample from normal distribution of size n
        sample = rand(Normal(), n)
        ybar[j] = mean(sample)
    end
    return ybar
end

# call the function r times and measure times:
n = 100
R = 10000
step_length = 100
reps = range(100, R, step=100)
runTimeJl = zeros(Int(R / step_length))

for i in eachindex(reps)
    t = @benchmark simMean($n, $reps[$i])
    runTimeJl[i] = mean(t).time / 1e9 # mean(t).time in nanoseconds
end

runtimeDf = DataFrame(runtime=runTimeJl)
CSV.write("Jlprocessed/01/Adv-Performance-Jl.csv", runtimeDf)

# import results (all in seconds):
```

```
R = CSV.read("Jlprocessed/01/Adv-Performance-R.csv", DataFrame)
Py = CSV.read("Jlprocessed/01/Adv-Performance-Py.csv", DataFrame)
Jl = CSV.read("Jlprocessed/01/Adv-Performance-Jl.csv", DataFrame)

# plot results
x_range = collect(reps)
plot(x_range, Jl.runtime, linestyle=:solid, color=:black,
    label="Julia", legend=:topleft)
plot!(x_range, R.runtime, linestyle=:dash, color=:black, label="R")
plot!(x_range, Py.runtime, linestyle=:dot, color=:black, label="Python")
xlabel!("Repetitions")
ylabel!("Runtime [seconds]")
savefig("JlGraphs/Adv-CompSpeed.pdf")


### Python-Code

# import random
# import numpy as np
# import timeit
# import pandas as pd
# import scipy.stats as stats

# np.random.seed(12345)

# def simMean(n, reps):
#     ybar = np.empty(reps)
#     for j in range(1, reps):
#         sample = stats.norm.rvs(0,1,size=n)
#         ybar[j] = np.mean(sample)
#     return(ybar)

# # call the function R times and measure times:
# n = 100
# R = 10000
# step_length = 100
# reps = range(100, R+1, step_length)
# runTime = np.empty(len(reps))
# number = 100

# for i in range(len(reps)):
#     runTime[i] = timeit.repeat(lambda: simMean(
#         n, reps[i]), number=100, repeat=1)[0] / number
# # repeat gives total time, i.e. number * single iteration time

# dfruntime = pd.DataFrame({"runtime": runTime})
# dfruntime.to_csv("Jlprocessed/01/Adv-Performance-Py.csv")


### R-Code

# library(microbenchmark)
# library(rio)

# set.seed(12345)

# simMean <- function(n, reps){
#   ybar <- numeric(reps)
#   for(j in 1:reps){
```

```
#     sample <- rnorm(n)
#     ybar[j] <- mean(sample)
#   }
#   return(ybar)
# }

# # call the function R times and measure times:
# n <- 100
# R <- 10000
# step_length <- 100
# reps <- seq(100, R, by=100)
# runtime <- numeric(R/step_length)


# for (i in 1:length(reps)){
#   t <- microbenchmark( simMean(n,reps[i]), unit = "s")
#   runtime[i] <- mean(t$time) / 1e9 # t$time in nanoseconds
# }

# export(as.data.frame(runtime),file="Jlprocessed/01/Adv-Performance-R.csv")
```

Script 1.52: `Simulate-Estimate.jl`

```
using Distributions, Random

# set the random seed:
Random.seed!(12345)

# set sample size:
n = 100

# draw a sample given the population parameters:
sample1 = rand(Normal(10, 2), n)

# estimate the population mean with the sample average:
estimate1 = mean(sample1)
println("estimate1 = $estimate1\n")

# draw a different sample and estimate again:
sample2 = rand(Normal(10, 2), n)
estimate2 = mean(sample2)
println("estimate2 = $estimate2\n")

# draw a third sample and estimate again:
sample3 = rand(Normal(10, 2), n)
estimate3 = mean(sample3)
print("estimate3: $estimate3")
```

Script 1.53: `Simulation-Repeated.jl`

```
using Distributions, Random

# set the random seed:
Random.seed!(12345)

# set sample size:
n = 100

# initialize ybar to an array of length r=10000 to later store results:
r = 10000
```

```
ybar = zeros(r)

# repeat r times:
for j in 1:r
    # draw a sample and store the sample mean in pos. j=1,... of ybar:
    sample = rand(Normal(10, 2), n)
    ybar[j] = mean(sample)
end
```

Script 1.54: Simulation-Repeated-Results.jl

```
using Distributions, Random, KernelDensity, Plots

# set the random seed:
Random.seed!(12345)

# set sample size:
n = 100

# initialize ybar to an array of length r=10000 to later store results:
r = 10000
ybar = zeros(r)

# repeat r times:
for j in 1:r
    # draw a sample and store the sample mean in pos. j=1,... of ybar:
    sample = rand(Normal(10, 2), n)
    ybar[j] = mean(sample)
end

# the first 8 of 10000 estimates:
ybar_preview = round.(ybar[1:8], digits=4)
println("ybar_preview: \n$ybar_preview\n")

# simulated mean:
mean_ybar = mean(ybar)
println("mean_ybar = $mean_ybar\n")

# simulated variance:
var_ybar = var(ybar)
println("var_ybar = $var_ybar")

# simulated density:
kde = KernelDensity.kde(ybar)

# normal density:
x_range = range(9, 11, length=100)
y = pdf.(Normal(10, sqrt(0.04)), x_range)

# create graph:
plot(kde.x, kde.density, color=:black, label="ybar", linewidth=2)
plot!(x_range, y, linestyle=:dash, color=:black,
    label="normal distribution", linewidth=2)
ylabel!("density")
xlabel!("ybar")
savefig("JlGraphs/Simulation-Repeated-Results.pdf")
```

Script 1.55: `Simulation-Inference-Figure.jl`

```
using Distributions, HypothesisTests, Plots, Random

# set the random seed:
Random.seed!(12345)

# set sample size and MC simulations:
r = 10000
n = 100

# initialize arrays to later store results:
CIlower = zeros(r)
CIupper = zeros(r)
pvalue1 = zeros(r)
pvalue2 = zeros(r)


# repeat r times:
for j in 1:r
    # draw a sample
    sample = rand(Normal(10, 2), n)
    sample_mean = mean(sample)
    sample_sd = std(sample)

    # test the (correct) null hypothesis mu=10:
    testres1 = OneSampleTTest(sample, 10)
    pvalue1[j] = pvalue(testres1)
    cv = quantile(TDist(n - 1), 0.975)
    CIlower[j] = sample_mean - cv * sample_sd / sqrt(n)
    CIupper[j] = sample_mean + cv * sample_sd / sqrt(n)

    # test the (incorrect) null hypothesis mu=9.5 & store the p value:
    testres2 = OneSampleTTest(sample, 9.5)
    pvalue2[j] = pvalue(testres2)

end

##################
## correct H0 ##
##################
plot(legend=false, size=(300, 500)) # initialize plot and set figure ratio
ylims!((0, 101))
xlims!((9, 11))
vline!([10], linestyle=:dash, color=:black, linewidth=0.5)
ylabel!("Sample No.")

for j in 1:100
    if 10 > CIlower[j] && 10 < CIupper[j]
        plot!([CIlower[j], CIupper[j]], [j, j], linestyle=:solid, color=:grey)
    else
        plot!([CIlower[j], CIupper[j]], [j, j], linestyle=:solid, color=:black)
    end
end
savefig("JlGraphs/Simulation-Inference-Figure1.pdf")

##################
## incorrect H0 ##
##################
plot(legend=false, size=(300, 500)) # initialize plot and set figure ratio
```

```
ylims!((0, 101))
xlims!((9, 11))
vline!([9.5], linestyle=:dash, color=:black, linewidth=0.5)
ylabel!("Sample No.")

for j in 1:100
    if 9.5 > CIlower[j] && 9.5 < CIupper[j]
        plot!([CIlower[j], CIupper[j]], [j, j], linestyle=:solid, color=:grey)
    else
        plot!([CIlower[j], CIupper[j]], [j, j], linestyle=:solid, color=:black)
    end
end
savefig("JlGraphs/Simulation-Inference-Figure2.pdf")
```

Script 1.56: Simulation-Inference.jl

```
using Distributions, Random, HypothesisTests

# set the random seed:
Random.seed!(12345)

# set sample size and MC simulations:
r = 10000
n = 100

# initialize arrays to later store results:
CIlower = zeros(r)
CIupper = zeros(r)
pvalue1 = zeros(r)
pvalue2 = zeros(r)


# repeat r times:
for j in 1:r
    # draw a sample
    sample = rand(Normal(10, 2), n)
    sample_mean = mean(sample)
    sample_sd = std(sample)

    # test the (correct) null hypothesis mu=10:
    testres1 = OneSampleTTest(sample, 10)
    pvalue1[j] = pvalue(testres1)
    cv = quantile(TDist(n - 1), 0.975)
    CIlower[j] = sample_mean - cv * sample_sd / sqrt(n)
    CIupper[j] = sample_mean + cv * sample_sd / sqrt(n)

    # test the (incorrect) null hypothesis mu=9.5 & store the p value:
    testres2 = OneSampleTTest(sample, 9.5)
    pvalue2[j] = pvalue(testres2)

end

# test results as logical value:
reject1 = pvalue1 .<= 0.05
count1_true = count(reject1) # counts true
count1_false = r - count1_true
println("count1_true: $count1_true\n")
println("count1_false: $count1_false\n")

reject2 = pvalue2 .<= 0.05
```

```
count2_true = count(reject2)
count2_false = r - count2_true
println("count2_true: $count2_true\n")
println("count2_false: $count2_false")
```

2. Scripts Used in Chapter 02

Script 2.1: `Example-2-3.jl`

```
using WooldridgeDatasets, DataFrames, Statistics

ceosal1 = DataFrame(wooldridge("ceosal1"))
x = ceosal1.roe
y = ceosal1.salary

# ingredients to the OLS formulas:
cov_xy = cov(x, y)
var_x = var(x)
x_bar = mean(x)
y_bar = mean(y)

# manual calculation of OLS coefficients:
b1 = cov_xy / var_x
b0 = y_bar - b1 * x_bar
println("b1 = $b1\n")
println("b0 = $b0")
```

Script 2.2: `Example-2-3-2.jl`

```
using WooldridgeDatasets, DataFrames, GLM

ceosal1 = DataFrame(wooldridge("ceosal1"))

reg = lm(@formula(salary ~ roe), ceosal1)
b = coef(reg)
println("b = $b")
```

Script 2.3: `Example-2-3-3.jl`

```
using WooldridgeDatasets, DataFrames, GLM, Plots

ceosal1 = DataFrame(wooldridge("ceosal1"))

reg = lm(@formula(salary ~ roe), ceosal1)

# scatter plot and fitted values:
fitted_values = predict(reg)
scatter(ceosal1.roe, ceosal1.salary, color=:grey80, label="observations")
plot!(ceosal1.roe, fitted_values, color=:black, linewidth=3, label="OLS")
xlabel!("roe")
ylabel!("salary")
savefig("JlGraphs/Example-2-3-3.pdf")

# instead of scatter, you can also use:
# plot(ceosal1.roe, ceosal1.salary, label="observations", seriestype=:scatter)
```

Script 2.4: `Example-2-4.jl`
```
using WooldridgeDatasets, DataFrames, GLM

wage1 = DataFrame(wooldridge("wage1"))

reg = lm(@formula(wage ~ educ), wage1)
b = coef(reg)
println("b = $b")
```

Script 2.5: `Example-2-5.jl`
```
using WooldridgeDatasets, DataFrames, GLM, Plots

vote1 = DataFrame(wooldridge("vote1"))

# OLS regression:
reg = lm(@formula(voteA ~ shareA), vote1)
b = coef(reg)
println("b = $b")

# scatter plot and fitted values:
fitted_values = predict(reg)
scatter(vote1.shareA, vote1.voteA,
    color=:grey, label="observations", legend=:topleft)
plot!(vote1.shareA, fitted_values, color=:black, linewidth=3, label="OLS")
xlabel!("shareA")
ylabel!("voteA")
savefig("JlGraphs/Example-2-5.pdf")
```

Script 2.6: `Example-2-6.jl`
```
using WooldridgeDatasets, DataFrames, GLM

ceosal1 = DataFrame(wooldridge("ceosal1"))

# OLS regression:
reg = lm(@formula(salary ~ roe), ceosal1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg\n")

# obtain predicted values and residuals:
salary_hat = predict(reg)
u_hat = residuals(reg)

# Wooldridge, Table 2.2:
table = DataFrame(roe=ceosal1.roe,
    salary=ceosal1.salary,
    salary_hat=salary_hat,
    u_hat=u_hat)
table_preview = first(table, 10)
println("table_preview: \n$table_preview")
```

Script 2.7: `Example-2-7.jl`
```
using WooldridgeDatasets, DataFrames, GLM, Statistics

wage1 = DataFrame(wooldridge("wage1"))
reg = lm(@formula(wage ~ educ), wage1)

# obtain coefficients, predicted values and residuals:
```

```
b = coef(reg)
wage_hat = predict(reg)
u_hat = residuals(reg)

# confirm property (1):
u_hat_mean = mean(u_hat)
println("u_hat_mean = $u_hat_mean\n")

# confirm property (2):
educ_u_cov = cov(wage1.educ, u_hat)
println("educ_u_cov = $educ_u_cov\n")

# confirm property (3):
educ_mean = mean(wage1.educ)
wage_pred = b[1] + b[2] * educ_mean
println("wage_pred = $wage_pred\n")

wage_mean = mean(wage1.wage)
println("wage_mean = $wage_mean")
```

Script 2.8: `Example-2-8.jl`

```
using WooldridgeDatasets, DataFrames, GLM, Statistics

ceosal1 = DataFrame(wooldridge("ceosal1"))

# OLS regression:
reg = lm(@formula(salary ~ roe), ceosal1)

# obtain predicted values and residuals:
sal_hat = predict(reg)
u_hat = residuals(reg)

# calculate R^2 in three different ways:
sal = ceosal1.salary
R2_a = var(sal_hat) / var(sal)
R2_b = 1 - var(u_hat) / var(sal)
R2_c = cor(sal, sal_hat)^2

println("R2_a = $R2_a\n")
println("R2_b = $R2_b\n")
println("R2_c = $R2_c")
```

Script 2.9: `Example-2-9.jl`

```
using WooldridgeDatasets, DataFrames, GLM

vote1 = DataFrame(wooldridge("vote1"))

# OLS regression:
reg = lm(@formula(voteA ~ shareA), vote1)

# print results using coeftable:
table_reg = coeftable(reg)
println("table_reg: \n$table_reg\n")

# accessing R^2:
r2_automatic = r2(reg)
println("r2_automatic = $r2_automatic")
```

Script 2.10: `Example-2-10.jl`

```
using WooldridgeDatasets, DataFrames, GLM

wage1 = DataFrame(wooldridge("wage1"))

# estimate log-level model:
reg = lm(@formula(log(wage) ~ educ), wage1)
b = coef(reg)
println("b = $b")
```

Script 2.11: `Example-2-11.jl`

```
using WooldridgeDatasets, DataFrames, GLM

ceosal1 = DataFrame(wooldridge("ceosal1"))

# estimate log-log model:
reg = lm(@formula(log(salary) ~ log(sales)), ceosal1)
b = coef(reg)
println("b = $b")
```

Script 2.12: `SLR-Origin-Const.jl`

```
using WooldridgeDatasets, DataFrames, GLM, Plots, Statistics

ceosal1 = DataFrame(wooldridge("ceosal1"))

# usual OLS regression:
reg1 = lm(@formula(salary ~ roe), ceosal1)
b1 = coef(reg1)
println("b1 = $b1\n")

# regression without intercept (through origin):
reg2 = lm(@formula(salary ~ 0 + roe), ceosal1)
b2 = coef(reg2)
println("b2 = $b2\n")

# regression without slope (on a constant):
reg3 = lm(@formula(salary ~ 1), ceosal1)
b3 = coef(reg3)
println("b3 = $b3\n")

# average y:
sal_mean = mean(ceosal1.salary)
println("sal_mean = $sal_mean")

# scatter plot and fitted values:
scatter(ceosal1.roe, ceosal1.salary, color="grey85", label="observations")
plot!(ceosal1.roe, predict(reg1), linewidth=2,
    color="black", label="full")
plot!(ceosal1.roe, predict(reg2), linewidth=2,
    color="dimgrey", label="trough origin")
plot!(ceosal1.roe, predict(reg3), linewidth=2,
    color="lightgrey", label="const only")
xlabel!("roe")
ylabel!("salary")
savefig("JlGraphs/SLR-Origin-Const.pdf")
```

Script 2.13: `Example-2-12.jl`

```
using WooldridgeDatasets, DataFrames, GLM, Statistics

meap93 = DataFrame(wooldridge("meap93"))

# estimate the model and save the results as reg:
reg = lm(@formula(math10 ~ lnchprg), meap93)

# number of obs.:
n = nobs(reg)

# SER:
u_hat_var = var(residuals(reg))
SER = sqrt(u_hat_var) * sqrt((n - 1) / (n - 2))
println("SER = $SER\n")

# SE of b0 and b1, respectively:
lnchprg_sq_mean = mean(meap93.lnchprg .^ 2)
lnchprg_var = var(meap93.lnchprg)
b0_se = SER / (sqrt(lnchprg_var) * sqrt(n - 1)) * sqrt(lnchprg_sq_mean)
b1_se = SER / (sqrt(lnchprg_var) * sqrt(n - 1))
println("b0_se = $b0_se\n")
println("b1_se = $b1_se\n")

# automatic calculations:
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Script 2.14: `SLR-Sim-Sample.jl`

```
using Random, GLM, DataFrames, Distributions, Statistics, Plots

# set the random seed:
Random.seed!(12345)

# set sample size:
n = 1000

# set true parameters (betas and sd of u):
beta0 = 1
beta1 = 0.5
su = 2

# draw a sample of size n:
x = rand(Normal(4, 1), n)
u = rand(Normal(0, su), n)
y = beta0 .+ beta1 .* x .+ u
df = DataFrame(y=y, x=x)

# estimate parameters by OLS:
reg = lm(@formula(y ~ x), df)
b = coef(reg)
println("b = $b\n")

# features of the sample for the variance formula:
x_sq_mean = mean(x .^ 2)
println("x_sq_mean = $x_sq_mean\n")
x_var = sum((x .- mean(x)) .^ 2)
println("x_var = $x_var")
```

```
# graph:
x_range = range(0, 8, length=100)
scatter(x, y, color="lightgrey", ylim=[-2, 10],
    label="sample", alpha=0.7, markerstrokecolor=:white)
plot!(x_range, beta0 .+ beta1 .* x_range, color="black",
    linestyle=:solid, linewidth=2, label="pop. regr. fct.")
plot!(x_range, coef(reg)[1] .+ coef(reg)[2] .* x_range, color="grey",
    linestyle=:solid, linewidth=2, label="OLS regr. fct.")
xlabel!("x")
ylabel!("y")
savefig("JlGraphs/SLR-Sim-Sample.pdf")
```

Script 2.15: SLR-Sim-Model.jl

```
using Random, GLM, DataFrames, Distributions, Statistics

# set the random seed:
Random.seed!(12345)

# set sample size and number of simulations:
n = 1000
r = 10000

# set true parameters (betas and sd of u):
beta0 = 1
beta1 = 0.5
su = 2
sx = 1
ex = 4

# initialize b0 and b1 to store results later:
b0 = zeros(r)
b1 = zeros(r)

# repeat r times:
for i in 1:r
        # draw a sample:
        x = rand(Normal(ex, sx), n)
        u = rand(Normal(0, su), n)
        y = beta0 .+ beta1 .* x .+ u
        df = DataFrame(y=y, x=x)
        # estimate OLS:
        reg = lm(@formula(y ~ x), df)
        b0[i] = coef(reg)[1]
        b1[i] = coef(reg)[2]
end
```

Script 2.16: SLR-Sim-Model-Condx.jl

```
using Random, GLM, DataFrames, Distributions, Statistics, Plots

# set the random seed:
Random.seed!(12345)

# set sample size and number of simulations:
n = 1000
r = 10000

# set true parameters (betas and sd of u):
beta0 = 1
```

```
beta1 = 0.5
su = 2

# initialize b0 and b1 to store results later:
b0 = zeros(r)
b1 = zeros(r)
# draw a sample of x, fixed over replications:
x = rand(Normal(4, 1), n)

# repeat r times:
for i in 1:r
        # draw a sample of y:
        u = rand(Normal(0, su), n)
        y = beta0 .+ beta1 .* x .+ u
        df = DataFrame(y=y, x=x)
        # estimate and store parameters by OLS:
        reg = lm(@formula(y ~ x), df)

        b0[i] = coef(reg)[1]
        b1[i] = coef(reg)[2]
end

# MC estimate of the expected values:
b0_mean = mean(b0)
b1_mean = mean(b1)
println("b0_mean = $b0_mean\n")
println("b1_mean = $b1_mean\n")

# MC estimate of the variances:
b0_var = var(b0)
b1_var = var(b1)
println("b0_var = $b0_var\n")
println("b1_var = $b1_var")

# graph:
x_range = range(0, 8, length=100)

# add population regression line:
plot(x_range, beta0 .+ beta1 .* x_range, ylim=[0, 6],
        color="black", linewidth=2, label="Population")

# add first OLS regression line (to attach a label):
plot!(x_range, b0[1] .+ b1[1] .* x_range,
        color="grey", linewidth=0.5, label="OLS regressions")

# add OLS regression lines no. 2 to 10:
for i in 2:10
        plot!(x_range, b0[i] .+ b1[i] .* x_range,
                color="grey", linewidth=0.5, label=false)
end
ylabel!("y")
xlabel!("x")
savefig("JlGraphs/SLR-Sim-Model-Condx.pdf")
```

Script 2.17: `SLR-Sim-Model-ViolSLR4.jl`

```
using Random, GLM, DataFrames, Distributions, Statistics

# set the random seed:
Random.seed!(12345)
```

```
# set sample size and number of simulations:
n = 1000
r = 10000

# set true parameters (betas and sd of u):
beta0 = 1
beta1 = 0.5
su = 2

# initialize b0 and b1 to store results later:
b0 = zeros(r)
b1 = zeros(r)
# draw a sample of x, fixed over replications:
x = rand(Normal(4, 1), n)

# repeat r times:
for i in 1:r
        # draw a sample of y:
        u_mean = (x .- 4) ./ 5
        u = rand.(Normal.(u_mean, su), 1)
        u = reduce(vcat, u)
        y = beta0 .+ beta1 .* x .+ u
        df = DataFrame(y=y, x=x)
        # estimate and store parameters by OLS:
        reg = lm(@formula(y ~ x), df)

        b0[i] = coef(reg)[1]
        b1[i] = coef(reg)[2]
end

# MC estimate of the expected values:
b0_mean = mean(b0)
b1_mean = mean(b1)
println("b0_mean = $b0_mean\n")
println("b1_mean = $b1_mean\n")

# MC estimate of the variances:
b0_var = var(b0)
b1_var = var(b1)
println("b0_var = $b0_var\n")
println("b1_var = $b1_var")
```

Script 2.18: SLR-Sim-Model-ViolSLR5.jl

```
using Random, GLM, DataFrames, Distributions, Statistics

# set the random seed:
Random.seed!(1234567)

# set sample size and number of simulations:
n = 1000
r = 10000

# set true parameters (betas and sd of u):
beta0 = 1
beta1 = 0.5
su = 2

# initialize b0 and b1 to store results later:
```

```
b0 = zeros(r)
b1 = zeros(r)
# draw a sample of x, fixed over replications:
x = rand(Normal(4, 1), n)

# repeat r times:
for i in 1:r
        # draw a sample of y:
        u_var = 4 / exp(4.5) .* exp.(x)
        u = rand.(Normal.(0, sqrt.(u_var)), 1)
        u = reduce(vcat, u)
        y = beta0 .+ beta1 .* x .+ u
        df = DataFrame(y=y, x=x)
        # estimate and store parameters by OLS:
        reg = lm(@formula(y ~ x), df)

        b0[i] = coef(reg)[1]
        b1[i] = coef(reg)[2]
end

# MC estimate of the expected values:
b0_mean = mean(b0)
b1_mean = mean(b1)
println("b0_mean = $b0_mean\n")
println("b1_mean = $b1_mean\n")

# MC estimate of the variances:
b0_var = var(b0)
b1_var = var(b1)
println("b0_var = $b0_var\n")
println("b1_var = $b1_var")
```

3. Scripts Used in Chapter 03

Script 3.1: `Example-3-1.jl`

```
using WooldridgeDatasets, GLM, DataFrames

gpa1 = DataFrame(wooldridge("gpa1"))

reg = lm(@formula(colGPA ~ hsGPA + ACT), gpa1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg\n")

r2_automatic = r2(reg)
println("r2_automatic = $r2_automatic")
```

Script 3.2: `Example-3-2.jl`

```
using WooldridgeDatasets, GLM, DataFrames

wage1 = DataFrame(wooldridge("wage1"))

reg = lm(@formula(log(wage) ~ educ + exper + tenure), wage1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Script 3.3: `Example-3-3.jl`

```
using WooldridgeDatasets, DataFrames, GLM

k401k = DataFrame(wooldridge("401k"))

reg = lm(@formula(prate ~ mrate + age), k401k)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Script 3.4: `Example-3-5a.jl`

```
using WooldridgeDatasets, DataFrames, GLM

crime1 = DataFrame(wooldridge("crime1"))

# model without avgsen:
reg = lm(@formula(narr86 ~ pcnv + ptime86 + qemp86), crime1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Script 3.5: `Example-3-5b.jl`

```
using WooldridgeDatasets, DataFrames, GLM

crime1 = DataFrame(wooldridge("crime1"))

# model with avgsen:
reg = lm(@formula(narr86 ~ pcnv + avgsen + ptime86 + qemp86), crime1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Script 3.6: `Example-3-6.jl`

```
using WooldridgeDatasets, DataFrames, GLM

wage1 = DataFrame(wooldridge("wage1"))

reg = lm(@formula(log(wage) ~ educ), wage1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Script 3.7: `OLS-Matrices.jl`

```
using WooldridgeDatasets, DataFrames, LinearAlgebra

gpa1 = DataFrame(wooldridge("gpa1"))

# determine sample size & no. of regressors:
n = size(gpa1)[1]
k = 2

# extract y:
y = gpa1.colGPA

# extract X and add a column of ones:
X = hcat(ones(n), gpa1.hsGPA, gpa1.ACT)
# display first rows of X:
X_preview = round.(X[1:3, :], digits=5)
println("X_preview = $X_preview\n")

# parameter estimates:
```

```
b = inv(transpose(X) * X) * transpose(X) * y
println("b = $b\n")

# residuals, estimated variance of u and SER:
u_hat = y - X * b
sigsq_hat = (transpose(u_hat) * u_hat) / (n - k - 1)
SER = sqrt(sigsq_hat)
println("SER = $SER\n")

# estimated variance of the parameter estimators and SE:
Vbeta_hat = sigsq_hat .* inv(transpose(X) * X)
se = sqrt.(diag(Vbeta_hat))
println("se = $se")
```

Script 3.8: `OLS-Matrices-Formula.jl`

```
using WooldridgeDatasets, DataFrames, StatsModels, LinearAlgebra
include("getMats.jl")

gpa1 = DataFrame(wooldridge("gpa1"))

# build y and X from a formula:
f = @formula(colGPA ~ 1 + hsGPA + ACT)
xy = getMats(f, gpa1)
y = xy[1]
X = xy[2]

# parameter estimates:
b = inv(transpose(X) * X) * transpose(X) * y
println("b = $b")
```

Script 3.9: `getMats.jl`

```
# for details, see https://juliastats.org/StatsModels.jl/stable/internals/
using StatsModels

function getMats(formula, df)
    f = apply_schema(formula, schema(formula, df))
    resp, pred = modelcols(f, df)
    return (resp, pred)
end
```

Script 3.10: `Omitted-Vars.jl`

```
using WooldridgeDatasets, DataFrames, GLM

gpa1 = DataFrame(wooldridge("gpa1"))

# parameter estimates for full and simple model:
reg = lm(@formula(colGPA ~ ACT + hsGPA), gpa1)
b = coef(reg)
println("b = $b\n")

# relation between regressors:
reg_delta = lm(@formula(hsGPA ~ ACT), gpa1)
delta_tilde = coef(reg_delta)
println("delta_tilde = $delta_tilde\n")

# omitted variables formula for b1_tilde:
b1_tilde = b[2] + b[3] * delta_tilde[2]
```

```
println("b1_tilde = $b1_tilde\n")

# actual regression with hsGPA omitted:
reg_om = lm(@formula(colGPA ~ ACT), gpa1)
b_om = coef(reg_om)
println("b_om = $b_om")
```

Script 3.11: `MLR-SE.jl`

```
using WooldridgeDatasets, DataFrames, GLM, Statistics

gpa1 = DataFrame(wooldridge("gpa1"))

# full estimation results including automatic SE:
reg = lm(@formula(colGPA ~ hsGPA + ACT), gpa1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg\n")

# calculation of SER via residuals:
n = nobs(reg)
k = length(coef(reg))
SER = sqrt(sum(residuals(reg) .^ 2) / (n - k))

# regressing hsGPA on ACT for calculation of R2 & VIF:
reg_hsGPA = lm(@formula(hsGPA ~ ACT), gpa1)
R2_hsGPA = r2(reg_hsGPA)
VIF_hsGPA = 1 / (1 - R2_hsGPA)
println("VIF_hsGPA = $VIF_hsGPA\n")

# manual calculation of SE of hsGPA coefficient:
sdx = std(gpa1.hsGPA) * sqrt((n - 1) / n)
SE_hsGPA = 1 / sqrt(n) * SER / sdx * sqrt(VIF_hsGPA)
println("SE_hsGPA = $SE_hsGPA")
```

4. Scripts Used in Chapter 04

Script 4.1: `Example-4-3-cv.jl`

```
using Distributions

# CV for alpha=5% and 1% using the t distribution with 137 d.f.:
alpha = [0.05, 0.01]
cv_t = round.(quantile.(TDist(137), 1 .- alpha ./ 2), digits=5)
println("cv_t = $cv_t\n")

# CV for alpha=5% and 1% using the normal approximation:
cv_n = round.(quantile.(Normal(), 1 .- alpha ./ 2), digits=5)
println("cv_n = $cv_n")
```

Script 4.2: `Example-4-3.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Distributions

gpa1 = DataFrame(wooldridge("gpa1"))

# store and display results:
reg = lm(@formula(colGPA ~ hsGPA + ACT + skipped), gpa1)
table_reg = coeftable(reg)
```

```
println("table_reg: \n$table_reg\n")

# manually confirm the formulas, i.e. extract coefficients and SE:
b = coef(reg)
se = stderror(reg)

# reproduce t statistic:
tstat = round.(b ./ se, digits=5)
println("tstat = $tstat\n")

# reproduce p value:
pval = round.(2 * cdf.(TDist(137), -abs.(tstat)), digits=5)
println("pval = $pval")
```

Script 4.3: **Example-4-1-cv.jl**

```
using Distributions

# CV for alpha=5% and 1% using the t distribution with 522 d.f.:
alpha = [0.05, 0.01]
cv_t = round.(quantile.(TDist(522), 1 .- alpha), digits=5)
println("cv_t = $cv_t\n")

# CV for alpha=5% and 1% using the normal approximation:
cv_n = round.(quantile.(Normal(), 1 .- alpha), digits=5)
println("cv_n = $cv_n")
```

Script 4.4: **Example-4-1.jl**

```
using WooldridgeDatasets, GLM, DataFrames

wage1 = DataFrame(wooldridge("wage1"))

reg = lm(@formula(log(wage) ~ educ + exper + tenure), wage1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Script 4.5: **Example-4-8.jl**

```
using WooldridgeDatasets, GLM, DataFrames, Distributions

rdchem = DataFrame(wooldridge("rdchem"))

# OLS regression:
reg = lm(@formula(log(rd) ~ log(sales) + profmarg), rdchem)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg\n")

# replicating 95% CI:
alpha = 0.05
CI95_upper = coef(reg) .+ stderror(reg) .* quantile(TDist(32 - 3), alpha / 2)
CI95_lower = coef(reg) .- stderror(reg) .* quantile(TDist(32 - 3), alpha / 2)
println("CI95_upper = $CI95_upper\n")
println("CI95_lower = $CI95_lower\n")

# calculating 99% CI:
alpha = 0.01
CI99_upper = coef(reg) .+ stderror(reg) .* quantile(TDist(32 - 3), alpha / 2)
CI99_lower = coef(reg) .- stderror(reg) .* quantile(TDist(32 - 3), alpha / 2)
println("CI99_upper = $CI99_upper\n")
println("CI99_lower = $CI99_lower")
```

Script 4.6: `F-Test.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Distributions

mlb1 = DataFrame(wooldridge("mlb1"))

# unrestricted OLS regression:
reg_ur = lm(@formula(log(salary) ~
        years + gamesyr + bavg + hrunsyr + rbisyr), mlb1)
r2_ur = r2(reg_ur)
println("r2_ur = $r2_ur\n")

# restricted OLS regression:
reg_r = lm(@formula(log(salary) ~ years + gamesyr), mlb1)
r2_r = r2(reg_r)
println("r2_r = $r2_r\n")

# F statistic:
n = nobs(reg_ur)
fstat = (r2_ur - r2_r) / (1 - r2_ur) * (n - 6) / 3
println("fstat = $fstat\n")

# CV for alpha=1% using the F distribution with 3 and 347 d.f.:
cv = quantile(FDist(3, 347), 1 - 0.01)
println("cv = $cv\n")

# p value = 1-cdf of the appropriate F distribution:
fpval = 1 - cdf(FDist(3, 347), fstat)
println("fpval = $fpval")
```

Script 4.7: `F-Test-Automatic.jl`

```
using WooldridgeDatasets, GLM, DataFrames

mlb1 = DataFrame(wooldridge("mlb1"))

# OLS regression:
reg_ur = lm(@formula(log(salary) ~
                years + gamesyr + bavg + hrunsyr + rbisyr), mlb1)
reg_r = lm(@formula(log(salary) ~
                years + gamesyr), mlb1)

# automated F test:
ftest_res = ftest(reg_r.model, reg_ur.model)

fstat = ftest_res.fstat[2]
fpval = ftest_res.pval[2]
println("fstat = $fstat\n")
println("fpval = $fpval")
```

Script 4.8: `F-Test-Automatic2.jl`

```
using WooldridgeDatasets, GLM, DataFrames

mlb1 = DataFrame(wooldridge("mlb1"))

# OLS regression:
reg_ur = lm(@formula(log(salary) ~
                years + gamesyr + bavg + hrunsyr + rbisyr), mlb1)

# restrictions "bavg = 0" and "hrunsyr = 2*rbisyr":
```

```
mlb1.newvar = 2 * mlb1.hrunsyr + mlb1.rbisyr
reg_r = lm(@formula(log(salary) ~ years + gamesyr + newvar), mlb1)

# automated F test:
ftest_res = ftest(reg_r.model, reg_ur.model)

fstat = ftest_res.fstat[2]
fpval = ftest_res.pval[2]
println("fstat = $fstat\n")
println("fpval = $fpval")
```

Script 4.9: `Example-4-8-2.jl`

```
using WooldridgeDatasets, GLM, DataFrames

rdchem = DataFrame(wooldridge("rdchem"))

# OLS regression:
reg_ur = lm(@formula(log(rd) ~ log(sales) + profmarg), rdchem)
reg_r = lm(@formula(log(rd) ~ 1), rdchem)

# automated F test:
ftest_res = ftest(reg_r.model, reg_ur.model)

fstat = ftest_res.fstat[2]
fpval = ftest_res.pval[2]
println("fstat = $fstat\n")
println("fpval = $fpval")
```

Script 4.10: `Example-4-10.jl`

```
using WooldridgeDatasets, GLM, DataFrames, RegressionTables

meap93 = DataFrame(wooldridge("meap93"))
meap93.b_s = meap93.benefits ./ meap93.salary

# estimate three different models:
reg1 = lm(@formula(log(salary) ~ b_s), meap93)
reg2 = lm(@formula(log(salary) ~ b_s + log(enroll) + log(staff)), meap93)
reg3 = lm(@formula(log(salary) ~
                b_s + log(enroll) + log(staff) + droprate + gradrate), meap93)

# print results with RegressionTables:
regtable(reg1, reg2, reg3)
```

5. Scripts Used in Chapter 05

Script 5.1: `Sim-Asy-OLS-norm.jl`

```
using Distributions, DataFrames, GLM, Random

# set the random seed:
Random.seed!(12345)

# set sample size and number of simulations:
n = 100
r = 10000
# set true parameters:
```

```
beta0 = 1
beta1 = 0.5
sx = 1
ex = 4
# initialize b1 to store results later:
b1 = zeros(r)
# draw a sample of x, fixed over replications:
x = rand(Normal(ex, sx), n)

# repeat r times:
for i in 1:r
    # draw a sample of u (std. normal):
    u = rand(Normal(0, 1), n)
    y = beta0 .+ beta1 .* x .+ u
    df = DataFrame(y=y, x=x)
    # estimate conditional OLS:
    reg = lm(@formula(y ~ x), df)
    b1[i] = coef(reg)[2]
end
```

Script 5.2: `Sim-Asy-OLS-chisq.jl`

```
using Distributions, DataFrames, GLM, Random

# set the random seed:
Random.seed!(12345)

# set sample size and number of simulations:
n = 100
r = 10000

# set true parameters:
beta0 = 1
beta1 = 0.5
sx = 1
ex = 4

# initialize b1 to store results later:
b1 = zeros(r)

# draw a sample of x, fixed over replications:
x = rand(Normal(ex, sx), n)

# repeat r times:
for i in 1:r
    # draw a sample of u (standardized chi-squared[1]):
    u = (rand(Chisq(1), n) .- 1) ./ sqrt(2)
    y = beta0 .+ beta1 .* x .+ u
    df = DataFrame(y=y, x=x)
    # estimate conditional OLS:
    reg = lm(@formula(y ~ x), df)
    b1[i] = coef(reg)[2]
end
```

Script 5.3: `Sim-Asy-OLS-uncond.jl`

```
using Distributions, DataFrames, GLM, Random

# set the random seed:
Random.seed!(12345)
```

```
# set sample size and number of simulations:
n = 100
r = 10000

# set true parameters:
beta0 = 1
beta1 = 0.5
sx = 1
ex = 4

# initialize b1 to store results later:
b1 = zeros(r)

# repeat r times:
for i in 1:r
    # draw a sample of x, varying over replications:
    x = rand(Normal(ex, sx), n)
    # draw a sample of u (std. normal):
    u = rand(Normal(0, 1), n)
    y = beta0 .+ beta1 .* x .+ u
    df = DataFrame(y=y, x=x)
    # estimate unconditional OLS:
    reg = lm(@formula(y ~ x), df)
    b1[i] = coef(reg)[2]
end
```

Script 5.4: `Example-5-3.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Distributions

crime1 = DataFrame(wooldridge("crime1"))

# 1. estimate restricted model:
reg_r = lm(@formula(narr86 ~ pcnv + ptime86 + qemp86), crime1)
r2_r = r2(reg_r)
println("r2_r = $r2_r\n")

# 2. regression of residuals from restricted model:
crime1.utilde = residuals(reg_r)
reg_LM = lm(@formula(utilde ~
                pcnv + ptime86 + qemp86 + avgsen + tottime), crime1)
r2_LM = r2(reg_LM)
println("r2_LM = $r2_LM\n")

# 3. calculation of LM test statistic:
LM = r2_LM * nobs(reg_LM)
println("LM = $LM\n")

# 4. critical value from chi-squared distribution, alpha=10%:
cv = quantile(Chisq(2), 1 - 0.10)
println("cv = $cv\n")

# 5. p value (alternative to critical value):
pval = 1 - cdf(Chisq(2), LM)
println("pval = $pval\n")

# 6. compare to F test:
reg_ur = lm(@formula(narr86 ~
                pcnv + ptime86 + qemp86 + avgsen + tottime), crime1)
```

```
# hypotheses: "avgsen = 0" and "tottime = 0"
reg_r = lm(@formula(narr86 ~ pcnv + ptime86 + qemp86), crime1)
ftest_res = ftest(reg_r.model, reg_ur.model)

fstat = ftest_res.fstat[2]
fpval = ftest_res.pval[2]
println("fstat = $fstat\n")
println("fpval = $fpval")
```

6. Scripts Used in Chapter 06

Script 6.1: `Data-Scaling.jl`

```
using WooldridgeDatasets, GLM, DataFrames, RegressionTables

bwght = DataFrame(wooldridge("bwght"))

# regress and report coefficients:
reg = lm(@formula(bwght ~ cigs + faminc), bwght)

# weight in pounds, manual way:
bwght.bwght_lbs = bwght.bwght ./ 16
reg_lbs1 = lm(@formula(bwght_lbs ~ cigs + faminc), bwght)

# weight in pounds, direct way:
reg_lbs2 = lm(@formula((bwght / 16) ~ cigs + faminc), bwght)

# packs of cigaretts:
reg_packs = lm(@formula(bwght ~ (cigs / 20) + faminc), bwght)

# weight in ounces using bwght_lbs:
reg_pds = lm(@formula(identity(bwght_lbs * 16) ~ cigs + faminc), bwght)

# print results with RegressionTables:
regtable(reg, reg_lbs1, reg_lbs2, reg_packs, reg_pds)
```

Script 6.2: `Example-6-1.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Statistics

hprice2 = DataFrame(wooldridge("hprice2"))

# define a function for the standardization:
function scale(x)
    x_mean = mean(x)
    x_var = var(x)
    x_scaled = (x .- x_mean) ./ sqrt.(x_var)
    return x_scaled
end

# standardize and estimate:
hprice2.price_sc = scale(hprice2.price)
hprice2.nox_sc = scale(hprice2.nox)
hprice2.crime_sc = scale(hprice2.crime)
hprice2.rooms_sc = scale(hprice2.rooms)
hprice2.dist_sc = scale(hprice2.dist)
hprice2.stratio_sc = scale(hprice2.stratio)
```

```
reg = lm(@formula(price_sc ~
        0 + nox_sc + crime_sc + rooms_sc + dist_sc + stratio_sc),
    hprice2)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Script 6.3: `Formula-Logarithm.jl`

```
using WooldridgeDatasets, GLM, DataFrames

hprice2 = DataFrame(wooldridge("hprice2"))

reg = lm(@formula(log(price) ~ log(nox) + rooms), hprice2)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Script 6.4: `Example-6-2.jl`

```
using WooldridgeDatasets, GLM, DataFrames

hprice2 = DataFrame(wooldridge("hprice2"))

reg = lm(@formula(log(price) ~
                log(nox) + log(dist) + rooms + (rooms^2) + stratio), hprice2)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Script 6.5: `Example-6-2-Ftest.jl`

```
using WooldridgeDatasets, GLM, DataFrames

hprice2 = DataFrame(wooldridge("hprice2"))

reg_ur = lm(@formula(log(price) ~
                log(nox) + log(dist) + rooms + (rooms^2) + stratio), hprice2)

# testing hypotheses rooms = 0 and rooms^2 = 0:
reg_r = lm(@formula(log(price) ~
                log(nox) + log(dist) + stratio), hprice2)

ftest_res = ftest(reg_r.model, reg_ur.model)
fstat = ftest_res.fstat[2]
fpval = ftest_res.pval[2]
println("fstat = $fstat\n")
println("fpval = $fpval")
```

Script 6.6: `Example-6-3.jl`

```
using WooldridgeDatasets, GLM, DataFrames

attend = DataFrame(wooldridge("attend"))

reg_ur = lm(@formula(stndfnl ~ atndrte * priGPA + ACT +
                               (priGPA^2) + (ACT^2)), attend)
table_reg_ur = coeftable(reg_ur)
println("table_reg_ur: \n$table_reg_ur\n")

# estimate for partial effect at priGPA=2.59:
b = coef(reg_ur)
partial_effect = b[2] + 2.59 * b[7]
println("partial_effect = $partial_effect\n")
```

```
# F test for partial effect at priGPA=2.59:
attend.pe = -2.59 .* attend.atndrte .+ attend.atndrte .* attend.priGPA
reg_r = lm(@formula(stndfnl ~ pe + priGPA + ACT +
                              (priGPA^2) + (ACT^2)), attend)
ftest_res = ftest(reg_r.model, reg_ur.model)
fstat = ftest_res.fstat[2]
fpval = ftest_res.pval[2]
println("fstat = $fstat\n")
println("fpval = $fpval")
```

Script 6.7: `Predictions.jl`

```
using WooldridgeDatasets, GLM, DataFrames

gpa2 = DataFrame(wooldridge("gpa2"))

reg = lm(@formula(colgpa ~ sat + hsperc + hsize + (hsize^2)), gpa2)

# print regression table:
table_reg = coeftable(reg)
println("table_reg: \n$table_reg\n")

# generate data set containing the regressor values for predictions:
cvalues1 = DataFrame(id="newPerson1", sat=1200, hsperc=30, hsize=5)
println("cvalues1: \n$cvalues1\n")

# point estimate of prediction (cvalues1):
colgpa_pred1 = round.(predict(reg, cvalues1), digits=5)
println("colgpa_pred1 = $colgpa_pred1\n")

# define three sets of regressor variables:
cvalues2 = DataFrame(id=["newPerson1", "newPerson2", "newPerson3"],
    sat=[1200, 900, 1400],
    hsperc=[30, 20, 5], hsize=[5, 3, 1])
println("cvalues2: \n$cvalues2\n")

# point estimate of prediction (cvalues2):
colgpa_pred2 = round.(predict(reg, cvalues2), digits=5)
println("colgpa_pred2 = $colgpa_pred2")
```

Script 6.8: `Example-6-5.jl`

```
using WooldridgeDatasets, GLM, DataFrames

gpa2 = DataFrame(wooldridge("gpa2"))

reg = lm(@formula(colgpa ~ sat + hsperc + hsize + (hsize^2)), gpa2)

# define three sets of regressor variables:
cvalues2 = DataFrame(
    id=["newPerson1", "newPerson2", "newPerson3"],
    sat=[1200, 900, 1400],
    hsperc=[30, 20, 5],
    hsize=[5, 3, 1])

# point estimates and 95% confidence and prediction intervals:
colgpa_CI_95 = predict(reg, cvalues2, interval=:confidence)
println("colgpa_CI_95: \n$colgpa_CI_95\n")
colgpa_PI_95 = predict(reg, cvalues2, interval=:prediction)
```

```
println("colgpa_PI_95: \n$colgpa_PI_95\n")

# point estimates and 99% confidence and prediction intervals:
colgpa_CI_99 = predict(reg, cvalues2, interval=:confidence, level=0.99)
println("colgpa_CI_99: \n$colgpa_CI_99\n")
colgpa_PI_99 = predict(reg, cvalues2, interval=:prediction, level=0.99)
println("colgpa_PI_99: \n$colgpa_PI_99")
```

Script 6.9: `Effects-Manual.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Plots, Statistics

hprice2 = DataFrame(wooldridge("hprice2"))

# repeating the regression from Example 6.2:
reg = lm(@formula(log(price) ~
        log(nox) + log(dist) + rooms + (rooms^2) + stratio), hprice2)

# predictions with rooms = 4-8, all others at the sample mean:
nox_mean = mean(hprice2.nox)
dist_mean = mean(hprice2.dist)
stratio_mean = mean(hprice2.stratio)
X = DataFrame(
    rooms=4:8,
    nox=nox_mean,
    dist=dist_mean,
    stratio=stratio_mean)
println("X: \n$X\n")

# calculate 95% confidence interval:
lpr_CI = predict(reg, X, interval=:confidence)
println("lpr_CI: \n$lpr_CI\n")

# plot:
plot(X.rooms, lpr_CI.prediction, color="black", label=false, legend=:topleft)
plot!(X.rooms, lpr_CI.upper, color="lightgrey", linestyle=:dash, label="upper CI")
plot!(X.rooms, lpr_CI.lower, color="darkgrey", linestyle=:dash, label="lower CI")
ylabel!("lprice")
xlabel!("rooms")
savefig("JlGraphs/Effects-Manual.pdf")
```

7. Scripts Used in Chapter 07

Script 7.1: `Example-7-1.jl`

```
using WooldridgeDatasets, GLM, DataFrames

wage1 = DataFrame(wooldridge("wage1"))

reg = lm(@formula(wage ~ female + educ + exper + tenure), wage1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Script 7.2: `Example-7-6.jl`

```
using WooldridgeDatasets, GLM, DataFrames

wage1 = DataFrame(wooldridge("wage1"))
```

```
reg = lm(@formula(log(wage) ~
        married * female + educ + exper + (exper^2) +
        tenure + (tenure^2)), wage1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Script 7.3: Example-7-1-Boolean.jl

```
using WooldridgeDatasets, GLM, DataFrames

wage1 = DataFrame(wooldridge("wage1"))

# regression with boolean variable:
wage1.isfemale = Bool.(wage1.female)
reg = lm(@formula(wage ~ isfemale + educ + exper + tenure), wage1,
    contrasts=Dict(:isfemale => DummyCoding()))

table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Script 7.4: Regr-Categorical.jl

```
using WooldridgeDatasets, GLM, DataFrames, FreqTables, CSV

CPS1985 = DataFrame(CSV.File("data/CPS1985.csv"))
# rename variable to make outputs more compact:
rename!(CPS1985, :occupation => :oc)

# table of categories and frequencies for two categorical variables:
freq_gender = freqtable(CPS1985.gender)
println("freq_gender: \n$freq_gender\n")

freq_occupation = freqtable(CPS1985.oc)
println("freq_occupation: \n$freq_occupation\n")

# directly using categorical variables in regression formula
# (the formula automatically interprets string
# columns as categorical variables and dummy codes them):
reg = lm(@formula(log(wage) ~ education + experience + gender + oc), CPS1985)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")

# rerun regression with different reference category:
reg_newref = lm(@formula(log(wage) ~ education + experience + gender + oc),
    CPS1985,
    contrasts=Dict(:gender => DummyCoding(base="male"),
        :oc => DummyCoding(base="technical")))
table_newref = coeftable(reg_newref)
println("table_newref: \n$table_newref")
```

Script 7.5: Example-7-8.jl

```
using WooldridgeDatasets, GLM, DataFrames, CategoricalArrays, FreqTables

lawsch85 = DataFrame(wooldridge("lawsch85"))

# define cut points for the rank:
cutpts = [1, 11, 26, 41, 61, 101, 176]
# note that "cut" takes intervals only in the form of [lower, upper)
```

```
# create categorical variable containing ranges for the rank:
lawsch85.rc = cut(lawsch85.rank, cutpts,
        labels=["[1,11)", "[11,26)", "[26,41)",
                "[41,61)", "[61,101)", "[101,176)"])

# display frequencies:
freq = freqtable(lawsch85.rc)
println("freq: \n$freq\n")

# run regression:
reg = lm(@formula(log(salary) ~ rc + LSAT + GPA + log(libvol) + log(cost)),
        lawsch85,
        contrasts=Dict(:rc => DummyCoding(base="[101,176)",
                levels=["[1,11)", "[11,26)", "[26,41)",
                        "[41,61)", "[61,101)", "[101,176)"])))
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Script 7.6: `Dummy-Interact.jl`

```
using WooldridgeDatasets, GLM, DataFrames

gpa3 = DataFrame(wooldridge("gpa3"))

# model with full interactions with female dummy (only for spring data):
reg_ur = lm(@formula(cumgpa ~ female * (sat + hsperc + tothrs)),
    subset(gpa3, :spring => ByRow(==(1))))
table_reg_ur = coeftable(reg_ur)
println("table_reg_ur: \n$table_reg_ur\n")

# F test for H0 (the interaction coefficients of "female" are zero):
reg_r = lm(@formula(cumgpa ~ sat + hsperc + tothrs),
    subset(gpa3, :spring => ByRow(==(1))))

ftest_res = ftest(reg_r.model, reg_ur.model)
fstat = ftest_res.fstat[2]
fpval = ftest_res.pval[2]
println("fstat = $fstat\n")
println("fpval = $fpval")
```

Script 7.7: `Dummy-Interact-Sep.jl`

```
using WooldridgeDatasets, GLM, DataFrames

gpa3 = DataFrame(wooldridge("gpa3"))

# estimate model for males (& spring data):
reg_m = lm(@formula(cumgpa ~ sat + hsperc + tothrs),
    subset(gpa3, :spring => ByRow(==(1)), :female => ByRow(==(0))))
table_reg_m = coeftable(reg_m)
println("table_reg_m: \n$table_reg_m")

# estimate model for females (& spring data):
reg_f = lm(@formula(cumgpa ~ sat + hsperc + tothrs),
    subset(gpa3, :spring => ByRow(==(1)), :female => ByRow(==(1))))
table_reg_f = coeftable(reg_f)
println("table_reg_f: \n$table_reg_f")
```

8. Scripts Used in Chapter 08

Script 8.1: `calc-white-se.jl`

```
using LinearAlgebra
include("../03/getMats.jl")

# for details, see Wooldridge (2010), p. 57
function calc_white_se(reg, df)
    f = formula(reg)
    xy = getMats(f, df)
    y = xy[1]
    X = xy[2]
    u = residuals(reg)
    invXX = inv(X' * X)
    sumterm = (X .* u)' * (X .* u)
    avar = invXX' * sumterm * invXX
    std_white = sqrt.(diag(avar))
    return std_white
end
```

Script 8.2: `Example-8-2-manual.jl`

```
using WooldridgeDatasets, GLM, DataFrames
include("../03/getMats.jl")
gpa3 = DataFrame(wooldridge("gpa3"))

reg_default = lm(@formula(cumgpa ~ sat + hsperc + tothrs +
                                   female + black + white),
        subset(gpa3, :spring => ByRow(==(1))))

# extract formula parts for SE calculation:
f = formula(reg_default)
xy = getMats(f, subset(gpa3, :spring => ByRow(==(1))))
y = xy[1]
X = xy[2]
u = residuals(reg_default)
df = DataFrame(X, :auto)


# calculate all rij:
ri1 = residuals(lm(@formula(x1 ~ 0 + x2 + x3 + x4 + x5 + x6 + x7), df))
ri2 = residuals(lm(@formula(x2 ~ 0 + x1 + x3 + x4 + x5 + x6 + x7), df))
ri3 = residuals(lm(@formula(x3 ~ 0 + x1 + x2 + x4 + x5 + x6 + x7), df))
ri4 = residuals(lm(@formula(x4 ~ 0 + x1 + x2 + x3 + x5 + x6 + x7), df))
ri5 = residuals(lm(@formula(x5 ~ 0 + x1 + x2 + x3 + x4 + x6 + x7), df))
ri6 = residuals(lm(@formula(x6 ~ 0 + x1 + x2 + x3 + x4 + x5 + x7), df))
ri7 = residuals(lm(@formula(x7 ~ 0 + x1 + x2 + x3 + x4 + x5 + x6), df))

# calculate SE according to Wooldridge (2019), Ch. 8.2:
se1 = sqrt(sum((ri1 .^ 2) .* (u .^ 2)) / (sum((ri1 .^ 2))^2))
se2 = sqrt(sum((ri2 .^ 2) .* (u .^ 2)) / (sum((ri2 .^ 2))^2))
se3 = sqrt(sum((ri3 .^ 2) .* (u .^ 2)) / (sum((ri3 .^ 2))^2))
se4 = sqrt(sum((ri4 .^ 2) .* (u .^ 2)) / (sum((ri4 .^ 2))^2))
se5 = sqrt(sum((ri5 .^ 2) .* (u .^ 2)) / (sum((ri5 .^ 2))^2))
se6 = sqrt(sum((ri6 .^ 2) .* (u .^ 2)) / (sum((ri6 .^ 2))^2))
se7 = sqrt(sum((ri7 .^ 2) .* (u .^ 2)) / (sum((ri7 .^ 2))^2))

se_white = round.([se1, se2, se3, se4, se5, se6, se7], digits=5)
println("se_white = $se_white")
```

Script 8.3: `Example-8-2.jl`

```
using WooldridgeDatasets, GLM, DataFrames
include("calc-white-se.jl")

gpa3 = DataFrame(wooldridge("gpa3"))

reg_default = lm(@formula(cumgpa ~ sat + hsperc + tothrs +
                                   female + black + white),
        subset(gpa3, :spring => ByRow(==(1))))

hc0 = calc_white_se(reg_default, subset(gpa3, :spring => ByRow(==(1))))

table_se = DataFrame(coefficients=coeftable(reg_default).rownms,
        b=round.(coef(reg_default), digits=5),
        se_default=round.(coeftable(reg_default).cols[2], digits=5),
        se_white=hc0)
println("table_se: \n$table_se")
```

Script 8.4: `Example-8-2-cont.jl`

```
using PyCall, WooldridgeDatasets, GLM, DataFrames
include("../03/getMats.jl")

# install Python's statsmodels with: using Conda; Conda.add("statsmodels")
sm = pyimport("statsmodels.api")

gpa3 = DataFrame(wooldridge("gpa3"))
gpa3_subset = subset(gpa3, :spring => ByRow(==(1)))

# F test using usual VCOV in Julia:
reg_ur = lm(@formula(cumgpa ~ sat + hsperc + tothrs + female + black + white),
    gpa3_subset)
reg_r = lm(@formula(cumgpa ~ sat + hsperc + tothrs + female),
    gpa3_subset)
ftest_res = ftest(reg_r.model, reg_ur.model)
fstat_jl = ftest_res.fstat[2]
fpval_jl = ftest_res.pval[2]
println("fstat_jl = $fstat_jl\n")
println("fpval_jl = $fpval_jl\n")

# F test using different variance-covariance formulas:
# definition of model and hypotheses:
f = @formula(cumgpa ~ 1 + sat + hsperc + tothrs + female + black + white)
xy = getMats(f, gpa3_subset)
reg = sm.OLS(xy[1], xy[2])
hypotheses = ["x5 = 0", "x6 = 0"] # meaning "black = 0" and "white = 0"

# usual VCOV in Python:
results_default = reg.fit()
ftest_py_default = results_default.f_test(hypotheses)
fstat_py_default = ftest_py_default.statistic
fpval_py_default = ftest_py_default.pvalue
println("fstat_py_default = $fstat_py_default\n")
println("fpval_py_default = $fpval_py_default\n")

# classical White VCOV in Python:
results_hc0 = reg.fit(cov_type="HC0")
ftest_py_hc0 = results_hc0.f_test(hypotheses)
fstat_py_hc0 = ftest_py_hc0.statistic
fpval_py_hc0 = ftest_py_hc0.pvalue
```

```
println("fstat_py_hc0 = $fstat_py_hc0\n")
println("fpval_py_hc0 = $fpval_py_hc0")
```

Script 8.5: `Example-8-4.jl`

```
using WooldridgeDatasets, GLM, DataFrames, HypothesisTests

hprice1 = DataFrame(wooldridge("hprice1"))

# estimate model:
f = @formula(price ~ 1 + lotsize + sqrft + bdrms)
reg = lm(f, hprice1)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg\n")

# automatic BP test (LM version),
# type = :linear specifies Breusch-Pagan test:
X = getMats(f, hprice1)[2]
result_bp_lm = WhiteTest(X, residuals(reg), type=:linear)
bp_lm_statistic = result_bp_lm.lm
bp_lm_pval = pvalue(result_bp_lm)
println("bp_lm_statistic = $bp_lm_statistic\n")
println("bp_lm_pval = $bp_lm_pval\n")

# manual BP test (F version):
hprice1.resid_sq = residuals(reg) .^ 2
reg_resid = lm(@formula(resid_sq ~ lotsize + sqrft + bdrms), hprice1)
bp_F = ftest(reg_resid.model)
bp_F_statistic = bp_F.fstat
bp_F_pval = bp_F.pval
println("bp_F_statistic = $bp_F_statistic\n")
println("bp_F_pval = $bp_F_pval")
```

Script 8.6: `Example-8-5.jl`

```
using WooldridgeDatasets, GLM, DataFrames, HypothesisTests
include("../03/getMats.jl")

hprice1 = DataFrame(wooldridge("hprice1"))

# estimate model:
f = @formula(log(price) ~ 1 + log(lotsize) + log(sqrft) + bdrms)
reg = lm(f, hprice1)

# BP test:
X = getMats(f, hprice1)[2]
result_bp = WhiteTest(X, residuals(reg), type=:linear)
bp_statistic = result_bp.lm
bp_pval = pvalue(result_bp)
println("bp_statistic = $bp_statistic\n")
println("bp_pval = $bp_pval\n")

# White test:
X_wh = hcat(ones(size(X)[1]),
    predict(reg),
    predict(reg) .^ 2)
result_white = WhiteTest(X_wh, residuals(reg), type=:linear)
white_statistic = result_white.lm
white_pval = pvalue(result_white)
```

```
println("white_statistic = $white_statistic\n")
println("white_pval = $white_pval")
```

Script 8.7: `Example-8-6.jl`

```
using WooldridgeDatasets, GLM, DataFrames
include("calc-white-se.jl")

k401ksubs = DataFrame(wooldridge("401ksubs"))

# subsetting data:
k401ksubs_sub = subset(k401ksubs, :fsize => ByRow(==(1)))

# OLS (only for singles, i.e. 'fsize'==1):
reg_ols = lm(@formula(nettfa ~ inc + ((age - 25)^2) + male + e401k),
        k401ksubs_sub)
hc0 = calc_white_se(reg_ols, k401ksubs_sub)

# print regression table with hc0:
table_ols = DataFrame(coefficients=coeftable(reg_ols).rownms,
        b=round.(coef(reg_ols), digits=5),
        se=round.(hc0, digits=5))
println("table_ols: \n$table_ols\n")

# WLS:
k401ksubs_sub.w = (1 ./ sqrt.(k401ksubs_sub.inc))
reg_wls = lm(@formula(identity(nettfa * w) ~ 0 + w + identity(inc * w) +
                                             identity((age - 25)^2 * w) +
                                             identity(male * w) +
                                             identity(e401k * w)), k401ksubs_sub)

# print regression table:
table_wls = DataFrame(coefficients=coeftable(reg_wls).rownms,
        b=round.(coef(reg_wls), digits=5),
        se=round.(coeftable(reg_wls).cols[2], digits=5))
println("table_wls: \n$table_wls\n")
```

Script 8.8: `WLS-Robust.jl`

```
using WooldridgeDatasets, GLM, DataFrames
include("calc-white-se.jl")

k401ksubs = DataFrame(wooldridge("401ksubs"))

# subsetting data:
k401ksubs_sub = subset(k401ksubs, :fsize => ByRow(==(1)))

# WLS:
k401ksubs_sub.w = (1 ./ sqrt.(k401ksubs_sub.inc))
reg_wls = lm(@formula(identity(nettfa * w) ~ 0 + w + identity(inc * w) +
                                             identity((age - 25)^2 * w) +
                                             identity(male * w) +
                                             identity(e401k * w)), k401ksubs_sub)

# robust results (White SE):
hc0 = calc_white_se(reg_wls, k401ksubs_sub)

# print regression table:
table_default = DataFrame(coefficients=coeftable(reg_wls).rownms,
        b=round.(coef(reg_wls), digits=5),
```

```
        se_default=round.(coeftable(reg_wls).cols[2], digits=5),
        se_robust=round.(hc0, digits=5))
println("table_default: \n$table_default")
```

Script 8.9: `Example-8-7.jl`

```
using WooldridgeDatasets, GLM, DataFrames, HypothesisTests

smoke = DataFrame(wooldridge("smoke"))

# OLS:
reg_ols = lm(@formula(cigs ~ log(income) + log(cigpric) +
                             educ + age + age^2 + restaurn), smoke)
table_ols = DataFrame(coefficients=coeftable(reg_ols).rownms,
        b=round.(coef(reg_ols), digits=5),
        se=round.(stderror(reg_ols), digits=5))
println("table_ols: \n$table_ols\n")

# BP test:
X = modelmatrix(reg_ols)
result_bp = WhiteTest(X, residuals(reg_ols), type=:linear)
bp_statistic = result_bp.lm
bp_pval = pvalue(result_bp)
println("bp_statistic = $bp_statistic\n")
println("bp_pval = $bp_pval\n")

# FGLS (estimation of the variance function):
smoke.logu2 = log.(residuals(reg_ols) .^ 2)
reg_fgls = lm(@formula(logu2 ~ log(income) + log(cigpric) +
                               educ + age + age^2 + restaurn), smoke)

table_fgls = DataFrame(coefficients=coeftable(reg_fgls).rownms,
        b=round.(coef(reg_fgls), digits=5),
        se=round.(stderror(reg_fgls), digits=5))
println("table_fgls: \n$table_fgls\n")

# FGLS (WLS):
smoke.w = (1 ./ sqrt.(exp.(predict(reg_fgls))))

reg_wls = lm(@formula(identity(cigs * w) ~ 0 + w + identity(log(income) * w) +
                                           identity(log(cigpric) * w) +
                                           identity(educ * w) +
                                           identity(age * w) +
                                           identity(age^2 * w) +
                                           identity(restaurn * w)), smoke)

table_wls = DataFrame(coefficients=coeftable(reg_wls).rownms,
        b=round.(coef(reg_wls), digits=5),
        se=round.(stderror(reg_wls), digits=5))
println("table_wls: \n$table_wls")
```

9. Scripts Used in Chapter 09

Script 9.1: `Example-9-2.jl`

```
using WooldridgeDatasets, GLM, DataFrames

hprice1 = DataFrame(wooldridge("hprice1"))
```

```
# original OLS:
reg = lm(@formula(price ~ lotsize + sqrft + bdrms), hprice1)

# regression for RESET test:
hprice1.fitted_sq = predict(reg) .^ 2 ./ 1000
hprice1.fitted_cub = predict(reg) .^ 3 ./ 1000

reg_reset = lm(@formula(price ~ lotsize + sqrft + bdrms +
                                fitted_sq + fitted_cub), hprice1)
table_reg_reset = coeftable(reg_reset)
println("table_reg_reset: \n$table_reg_reset\n")

# RESET test (H0: all coefficients including "fitted" are zero):
ftest_res = ftest(reg.model, reg_reset.model)
fstat = ftest_res.fstat[2]
fpval = ftest_res.pval[2]
println("fstat = $fstat\n")
println("fpval = $fpval")
```

Script 9.2: `Nonnested-Test.jl`

```
using WooldridgeDatasets, GLM, DataFrames

hprice1 = DataFrame(wooldridge("hprice1"))

# two alternative models:
reg1 = lm(@formula(price ~ lotsize + sqrft + bdrms), hprice1)
reg2 = lm(@formula(price ~ log(lotsize) + log(sqrft) + bdrms), hprice1)

# encompassing test of Davidson & MacKinnon:
# comprehensive model:
reg3 = lm(@formula(price ~ lotsize + sqrft + bdrms +
                           log(lotsize) + log(sqrft)), hprice1)

# test model 1:
ftest_res1 = ftest(reg1.model, reg3.model)

fstat1 = ftest_res1.fstat[2]
fpval1 = ftest_res1.pval[2]
println("fstat1 = $fstat1\n")
println("fpval1 = $fpval1\n")

# test model 2:
ftest_res2 = ftest(reg2.model, reg3.model)

fstat2 = ftest_res2.fstat[2]
fpval2 = ftest_res2.pval[2]
println("fstat2 = $fstat2\n")
println("fpval2 = $fpval2")
```

Script 9.3: `Sim-ME-Dep.jl`

```
using Random, Distributions, Statistics, GLM, DataFrames

# set the random seed:
Random.seed!(12345)

# set sample size and number of simulations:
n = 1000
```

```
r = 10000

# set true parameters (betas):
beta0 = 1
beta1 = 0.5

# initialize arrays to store results later (b1 without ME, b1_me with ME):
b1 = zeros(r)
b1_me = zeros(r)

# draw a sample of x, fixed over replications:
x = rand(Normal(4, 1), n)

# repeat r times:
for i in 1:r
        # draw a sample of u:
        u = rand(Normal(0, 1), n)

        # draw a sample of ystar:
        ystar = beta0 .+ beta1 * x .+ u

        # measurement error and mismeasured y:
        e0 = rand(Normal(0, 1), n)
        y = ystar .+ e0
        df = DataFrame(ystar=ystar, y=y, x=x)

        # regress ystar on x and store slope estimate at position i:
        reg_star = lm(@formula(ystar ~ x), df)
        b1[i] = coef(reg_star)[2]

        # regress y on x and store slope estimate at position i:
        reg_me = lm(@formula(y ~ x), df)
        b1_me[i] = coef(reg_me)[2]
end

# mean with and without ME:
b1_mean = mean(b1)
b1_me_mean = mean(b1_me)
println("b1_mean = $b1_mean\n")
println("b1_me_mean = $b1_me_mean\n")

# variance with and without ME:
b1_var = var(b1)
b1_me_var = var(b1_me)
println("b1_var = $b1_var\n")
println("b1_me_var = $b1_me_var")
```

Script 9.4: `Sim-ME-Explan.jl`

```
using Random, Distributions, Statistics, GLM, DataFrames

# set the random seed:
Random.seed!(12345)

# set sample size and number of simulations:
n = 1000
r = 10000

# set true parameters (betas):
beta0 = 1
```

```
beta1 = 0.5

# initialize arrays to store results later (b1 without ME, b1_me with ME):
b1 = zeros(r)
b1_me = zeros(r)

# draw a sample of x, fixed over replications:
xstar = rand(Normal(4, 1), n)

# repeat r times:
for i in 1:r
        # draw a sample of u:
        u = rand(Normal(0, 1), n)

        # draw a sample of y:
        y = beta0 .+ beta1 * xstar .+ u

        # measurement error and mismeasured x:
        e1 = rand(Normal(0, 1), n)
        x = xstar .+ e1
        df = DataFrame(y=y, xstar=xstar, x=x)

        # regress y on xstar and store slope estimate at position i:
        reg_star = lm(@formula(y ~ xstar), df)
        b1[i] = coef(reg_star)[2]

        # regress y on x and store slope estimate at position i:
        reg_me = lm(@formula(y ~ x), df)
        b1_me[i] = coef(reg_me)[2]
end

# mean with and without ME:
b1_mean = mean(b1)
b1_me_mean = mean(b1_me)
println("b1_mean = $b1_mean\n")
println("b1_me_mean = $b1_me_mean\n")

# variance with and without ME:
b1_var = var(b1)
b1_me_var = var(b1_me)
println("b1_var = $b1_var\n")
println("b1_me_var = $b1_me_var")
```

Script 9.5: `NaN-Inf-Missing.jl`

```
using Distributions, DataFrames, Statistics

# NaN, missings and infinite values in Julia:
x1 = [0, 2, NaN, Inf, missing]
logx = log.(x1)
invx = 0 ./ x1
isnanx = isnan.(x1)
isinfx = isinf.(x1)
ismissingx = ismissing.(x1)

results = DataFrame(x1=x1, logx=logx, invx=invx, ismissingx=ismissingx,
    isnanx=isnanx, isinfx=isinfx)
println("results = $results\n")

# mathematically not defined is not always NaN (like in R or Python):
```

```
test = try
    log(-1) # results in an ERROR
catch e
    NaN
end
println("test = $test\n")

# handling missings:
x2 = [4, 2, missing, 3]
meanx2_1 = mean(x2)
println("meanx2_1 = $meanx2_1\n")

meanx2_2 = mean(skipmissing(x2))
println("meanx2_2 = $meanx2_2\n")

x3 = [4, 2, NaN, 3]
meanx3_1 = mean(x3)
println("meanx3_1 = $meanx3_1\n")

meanx3_2 = mean(x3[.!isnan.(x3)])
println("meanx3_2 = $meanx3_2")
```

Script 9.6: `Missings.jl`

```
using WooldridgeDatasets, GLM, DataFrames

lawsch85 = DataFrame(wooldridge("lawsch85"))
lsat = lawsch85.LSAT

# create boolean indicator for missings:
missLSAT = ismissing.(lsat)

# LSAT and indicator for Schools No. 120-129:
preview = DataFrame(lsat=lsat[120:129],
    missLSAT=missLSAT[120:129])
println("preview: \n$preview\n")

# frequencies of indicator:
tot_missing = count(missLSAT) # same as sum(missLSAT)
tot_nonmissings = count(.!missLSAT)
println("tot_missing = $tot_missing\n")
println("tot_nonmissings = $tot_nonmissings\n")

# missings for all variables in data frame (counts):
miss_all = ismissing.(lawsch85)
freq_missLSAT = mapcols(count, miss_all)
freq_missLSAT_preview = freq_missLSAT[:, 1:9] # print only first nine columns
println("freq_missLSAT_preview: \n$freq_missLSAT_preview\n")

# computing amount of complete cases:
lsat_compl_cases1 = dropmissing(lawsch85)
complete_cases1 = nrow(lsat_compl_cases1)
println("complete_cases1 = $complete_cases1\n")

lsat_compl_cases2 = completecases(lawsch85)
complete_cases2 = count(lsat_compl_cases2)
println("complete_cases2 = $complete_cases2")
```

Script 9.7: `Missings-Analyses.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Statistics

lawsch85 = DataFrame(wooldridge("lawsch85"))

# missings:
x = lawsch85.LSAT
x_bar1 = mean(x)
x_bar2 = mean(skipmissing(x))
println("x_bar1 = $x_bar1\n")
println("x_bar2 = $x_bar2\n")

# observations and variables:
nrows = nrow(lawsch85)
ncols = ncol(lawsch85)
println("nrows = $nrows\n")
println("ncols = $ncols\n")

# regression (missings are taken care of by default):
reg = lm(@formula(log(salary) ~ LSAT + cost + age), lawsch85)
n = nobs(reg)
println("n = $n")
```

Script 9.8: `Outliers.jl`

```
using WooldridgeDatasets, GLM, DataFrames, LinearAlgebra, Plots

rdchem = DataFrame(wooldridge("rdchem"))

# create dummys for each observation with an identity matrix:
n = nrow(rdchem)
dummys = DataFrame(Matrix(1I, n, n), Symbol.(:d, 1:n)) # colnames d1, ..., d32

# studentized residuals for all observations:
studres = zeros(n)
for i in 1:n
    rdchem.di = dummys[:, i]
    reg_i = lm(@formula(rdintens ~ sales + profmarg + di), rdchem)
    # save t statistic (3rd column) of di (4th element):
    studres[i] = coeftable(reg_i).cols[3][4]
end

# display extreme values:
studres_max = maximum(studres)
studres_min = minimum(studres)
println("studres_max = $studres_max\n")
println("studres_min = $studres_min")

# histogram:
histogram(studres, color="grey", legend=false)
xlabel!("studres")
savefig("JlGraphs/Outliers.pdf")
```

Script 9.9: `LAD.jl`

```
using WooldridgeDatasets, DataFrames, GLM, QuantileRegressions

rdchem = DataFrame(wooldridge("rdchem"))

# OLS regression:
```

```
reg_ols = lm(@formula(rdintens ~ sales / 1000 + profmarg), rdchem)
table_reg_ols = coeftable(reg_ols)
println("table_reg_ols: \n$table_reg_ols\n")

# LAD regression:
reg_lad = qreg(@formula(rdintens ~ sales / 1000 + profmarg), rdchem, 0.5)
table_reg_lad = coeftable(reg_lad)
println("table_reg_lad: \n$table_reg_lad")
```

10. Scripts Used in Chapter 10

Script 10.1: Example-10-2.jl

```
using WooldridgeDatasets, GLM, DataFrames

intdef = DataFrame(wooldridge("intdef"))

# linear regression of static model:
reg = lm(@formula(i3 ~ inf + def), intdef)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Script 10.2: Example-Barium.jl

```
using WooldridgeDatasets, DataFrames, Dates, Plots

barium = DataFrame(wooldridge("barium"))
T = nrow(barium)

# monthly time series starting Feb. 1978:
barium.date = range(Date(1978, 2, 1), step=Month(1), length=T)
preview = barium[1:5, ["date", "chnimp"]]
println("preview: \n$preview")

# plot chnimp:
plot(barium.date, barium.chnimp, legend=false, color="grey")
ylabel!("chnimp")
savefig("JlGraphs/Example-Barium.pdf")
```

Script 10.3: Example-StockData.jl

```
using DataFrames, Dates, MarketData, Plots

# download data for "F" (= Ford Motor Company) and define start and end:
ticker = "F"
start_date = DateTime(2014, 1, 1)
end_date = DateTime(2016, 1, 1)

# import data:
F_data = yahoo(ticker, YahooOpt(period1=start_date, period2=end_date))

# look at imported data:
F_data_head = first(DataFrame(F_data), 5)
println("F_data_head: \n$F_data_head\n")

F_data_tail = last(DataFrame(F_data), 5)
println("F_data_tail: \n$F_data_tail")
```

```
# time series plot of adjusted closing prices:
plot(F_data.AdjClose, legend=false, color="grey")
ylabel!("AdjClose")
savefig("JlGraphs/Example-StockData.pdf")
```

Script 10.4: `Example-10-4.jl`

```
using WooldridgeDatasets, GLM, DataFrames

fertil3 = DataFrame(wooldridge("fertil3"))

# add all lags of pe up to order 2:
fertil3.pe_lag1 = lag(fertil3.pe, 1)
fertil3.pe_lag2 = lag(fertil3.pe, 2)

# linear regression of model with lags:
reg = lm(@formula(gfr ~ pe + pe_lag1 + pe_lag2 + ww2 + pill), fertil3)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Script 10.5: `Example-10-4-cont.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Distributions

fertil3 = DataFrame(wooldridge("fertil3"))

# add all lags of pe up to order 2:
fertil3.pe_lag1 = lag(fertil3.pe, 1)
fertil3.pe_lag2 = lag(fertil3.pe, 2)

# handle missings due to lagged data manually (important for ftest):
fertil3 = fertil3[Not([1, 2]), :]

# linear regression of model with lags:
reg_ur = lm(@formula(gfr ~ pe + pe_lag1 + pe_lag2 + ww2 + pill), fertil3)

# F test (H0: all pe coefficients are zero):
reg_r = lm(@formula(gfr ~ ww2 + pill), fertil3)
ftest_res = ftest(reg_r.model, reg_ur.model)
fstat = ftest_res.fstat[2]
fpval = ftest_res.pval[2]
println("fstat = $fstat\n")
println("fpval = $fpval\n")

# calculating the LRP:
b_pe_tot = sum(coef(reg_ur)[[2, 3, 4]])
println("b_pe_tot = $b_pe_tot\n")

# testing LRP=0:
fertil3.ptm1pt = fertil3.pe_lag1 - fertil3.pe
fertil3.ptm2pt = fertil3.pe_lag2 - fertil3.pe
reg_LRP = lm(@formula(gfr ~ pe + ptm1pt + ptm2pt + ww2 + pill), fertil3)
table_res_LRP = coeftable(reg_LRP)
println("table_res_LRP: \n$table_res_LRP")
```

Script 10.6: `Example-10-7.jl`

```
using WooldridgeDatasets, GLM, DataFrames

hseinv = DataFrame(wooldridge("hseinv"))
```

```
# linear regression without time trend:
reg_wot = lm(@formula(log(invpc) ~ log(price)), hseinv)
table_reg_wot = coeftable(reg_wot)
println("table_reg_wot: \n$table_reg_wot\n")

# linear regression with time trend (data set includes a time variable t):
reg_wt = lm(@formula(log(invpc) ~ log(price) + t), hseinv)
table_reg_wt = coeftable(reg_wt)
println("table_reg_wt: \n$table_reg_wt")
```

Script 10.7: `Example-10-11.jl`

```
using WooldridgeDatasets, GLM, DataFrames

barium = DataFrame(wooldridge("barium"))

# linear regression with seasonal effects:
reg = lm(@formula(log(chnimp) ~ log(chempi) + log(gas) +
                                log(rtwex) + befile6 + affile6 + afdec6 +
                                feb + mar + apr + may + jun + jul +
                                aug + sep + oct + nov + dec), barium)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

11. Scripts Used in Chapter 11

Script 11.1: `Example-11-4.jl`

```
using WooldridgeDatasets, GLM, DataFrames, RegressionTables

nyse = DataFrame(wooldridge("nyse"))
nyse.ret = nyse.return

# add all lags up to order 3:
nyse.ret_lag1 = lag(nyse.ret, 1)
nyse.ret_lag2 = lag(nyse.ret, 2)
nyse.ret_lag3 = lag(nyse.ret, 3)

# linear regression of model with lags:
reg1 = lm(@formula(ret ~ ret_lag1), nyse)
reg2 = lm(@formula(ret ~ ret_lag1 + ret_lag2), nyse)
reg3 = lm(@formula(ret ~ ret_lag1 + ret_lag2 + ret_lag3), nyse)

# print results with RegressionTables:
regtable(reg1, reg2, reg3)
```

Script 11.2: `Example-EffMkts.jl`

```
using DataFrames, GLM, Dates, MarketData, Plots, RegressionTables

# download data for "AAPL" (= Apple) and define start and end:
ticker = "AAPL"
start_date = DateTime(2007, 12, 31)
end_date = DateTime(2017, 01, 01)

# import data as DataFrame:
AAPL_data = DataFrame(yahoo(ticker,
```

```
    YahooOpt(period1=start_date, period2=end_date)))

# calculate return as the difference of logged prices:
AAPL_data.ret = vcat(missing, diff(log.(AAPL_data.AdjClose)))

# time series plot of returns:
plot(AAPL_data.timestamp, AAPL_data.ret, legend=false, color="grey")
ylabel!("returns")
savefig("JlGraphs/Example-EffMkts.pdf")

# linear regression of models with lags:
AAPL_data.ret_lag1 = lag(AAPL_data.ret, 1)
AAPL_data.ret_lag2 = lag(AAPL_data.ret, 2)
AAPL_data.ret_lag3 = lag(AAPL_data.ret, 3)

reg1 = lm(@formula(ret ~ ret_lag1), AAPL_data)
reg2 = lm(@formula(ret ~ ret_lag1 + ret_lag2), AAPL_data)
reg3 = lm(@formula(ret ~ ret_lag1 + ret_lag2 + ret_lag3), AAPL_data)

# print results with RegressionTables:
regtable(reg1, reg2, reg3)
```

Script 11.3: `Simulate-RandomWalk.jl`

```
using Random, Distributions, Statistics, Plots

# set the random seed:
Random.seed!(12345)

# initialize plot:
x_range = range(0, 50, 51)
plot(xlims=(0, 50), ylims=(-25, 25))

# loop over draws:
for r in 1:30
    # i.i.d. standard normal shock:
    e = rand(Normal(0, 1), 51)

    # set first entry to 0 (gives y_0 = 0):
    e[1] = 0

    # random walk as cumulative sum of shocks:
    y = cumsum(e)

    # add line to graph:
    plot!(x_range, y, color="lightgrey", legend=false)
end

hline!([0], color="black", linewidth=2, linestyle=:dash)
xlabel!("time")
ylabel!("y")
savefig("JlGraphs/Simulate-RandomWalk.pdf")
```

Script 11.4: `Simulate-RandomWalkDrift.jl`

```
using Random, Distributions, Statistics, Plots

# set the random seed:
Random.seed!(12345)
```

```
# initialize plot:
x_range = range(0, 50, 51)
plot(xlims=(0, 50), ylims=(0, 100))

# loop over draws:
for r in 1:30
    # i.i.d. standard normal shock:
    e = rand(Normal(0, 1), 51)

    # set first entry to 0 (gives y_0 = 0):
    e[1] = 0

    # random walk as cumulative sum of shocks:
    y = cumsum(e) + 2 * x_range

    # add line to graph:
    plot!(x_range, y, color="lightgrey", legend=false)
end

plot!(x_range, 2 * x_range, color="black", linewidth=2, linestyle=:dash)
xlabel!("time")
ylabel!("y")
savefig("JlGraphs/Simulate-RandomWalkDrift.pdf")
```

Script 11.5: `Simulate-RandomWalkDrift-Diff.jl`

```
using Random, Distributions, Statistics, Plots

# set the random seed:
Random.seed!(12345)

# initialize plot:
x_range = range(1, 50, 50)
plot(xlims=(0, 50), ylims=(-1, 5))

# loop over draws:
for r in 1:30
    # i.i.d. standard normal shock:
    e = rand(Normal(0, 1), 51)

    # set first entry to 0 (gives y_0 = 0):
    e[1] = 0

    # random walk as cumulative sum of shocks:
    y = cumsum(2 .+ e)

    # first difference:
    Dy = y[2:51] .- y[1:50]

    # add line to graph:
    plot!(x_range, Dy, color="lightgrey", legend=false)
end

hline!([2], color="black", linewidth=2, linestyle=:dash)
xlabel!("time")
ylabel!("Dy")
savefig("JlGraphs/Simulate-RandomWalkDrift-Diff.pdf")
```

Script 11.6: `Example-11-6.jl`

```
using WooldridgeDatasets, GLM, DataFrames

fertil3 = DataFrame(wooldridge("fertil3"))

# compute first differences (first difference is always missing):
fertil3.gfr_diff1 = vcat(missing, diff(fertil3.gfr))
fertil3.pe_diff1 = vcat(missing, diff(fertil3.pe))
preview = fertil3[1:5, ["gfr", "gfr_diff1", "pe", "pe_diff1"]]
println("preview: \n$preview\n")

# linear regression of model with first differences:
reg1 = lm(@formula(gfr_diff1 ~ pe_diff1), fertil3)
table_reg1 = coeftable(reg1)
println("table_reg1: \n$table_reg1\n")

# linear regression of model with lagged differences:
fertil3.pe_diff1_lag1 = lag(fertil3.pe_diff1, 1)
fertil3.pe_diff1_lag2 = lag(fertil3.pe_diff1, 2)

reg2 = lm(@formula(gfr_diff1 ~ pe_diff1 + pe_diff1_lag1 + pe_diff1_lag2),
    fertil3)
table_reg2 = coeftable(reg2)
println("table_reg2: \n$table_reg2")
```

12. Scripts Used in Chapter 12

Script 12.1: `Example-12-2-Static.jl`

```
using WooldridgeDatasets, GLM, DataFrames

phillips = DataFrame(wooldridge("phillips"))
yt96 = subset(phillips, :year => ByRow(<=(1996)))

# estimation of static Phillips curve:
reg_s = lm(@formula(inf ~ unem), yt96)

# residuals and AR(1) test:
yt96.resid_s = residuals(reg_s)
yt96.resid_s_lag1 = lag(yt96.resid_s, 1)

reg = lm(@formula(resid_s ~ resid_s_lag1), yt96)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Script 12.2: `Example-12-2-ExpAug.jl`

```
using WooldridgeDatasets, GLM, DataFrames

phillips = DataFrame(wooldridge("phillips"))
yt96 = subset(phillips, :year => ByRow(<=(1996)))

# estimation of expectations-augmented Phillips curve:
yt96.inf_diff1 = vcat(missing, diff(yt96.inf))
yt96 = yt96[Not(1), :]
reg_ea = lm(@formula(inf_diff1 ~ unem), yt96)

yt96.resid_ea = residuals(reg_ea)
```

```
yt96.resid_ea_lag1 = lag(yt96.resid_ea, 1)

reg = lm(@formula(resid_ea ~ resid_ea_lag1), yt96)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Script 12.3: `Example-12-4.jl`

```
using WooldridgeDatasets, GLM, DataFrames

barium = DataFrame(wooldridge("barium"))

reg = lm(@formula(log(chnimp) ~ log(chempi) + log(gas) + log(rtwex) +
                                befile6 + affile6 + afdec6), barium)

# testing resid_lag1 = 0, resid_lag2 = 0 and resid_lag3 = 0:
barium.resid = residuals(reg)
barium.resid_lag1 = lag(barium.resid, 1)
barium.resid_lag2 = lag(barium.resid, 2)
barium.resid_lag3 = lag(barium.resid, 3)
barium = barium[Not(1:3), :]

reg_manual_ur = lm(@formula(resid ~ resid_lag1 + resid_lag2 + resid_lag3 +
                                    log(chempi) + log(gas) + log(rtwex) +
                                    befile6 + affile6 + afdec6), barium)
reg_manual_r = lm(@formula(resid ~ log(chempi) + log(gas) + log(rtwex) +
                                   befile6 + affile6 + afdec6), barium)
ftest_manual_res = ftest(reg_manual_r.model, reg_manual_ur.model)

fstat_manual = ftest_manual_res.fstat[2]
fpval_manual = ftest_manual_res.pval[2]
println("fstat_manual = $fstat_manual\n")
println("fpval_manual = $fpval_manual")
```

Script 12.4: `Example-DWtest.jl`

```
using WooldridgeDatasets, GLM, DataFrames, HypothesisTests
include("../03/getMats.jl")

phillips = DataFrame(wooldridge("phillips"))
yt96 = subset(phillips, :year => ByRow(<=(1996)))

# estimation of both Phillips curve models and
# extraction of regressor matrices and residuals:
reg_s = lm(@formula(inf ~ unem), yt96)
X_s = getMats(formula(reg_s), yt96)[2]
resid_s = residuals(reg_s)

yt96.inf_diff1 = vcat(missing, diff(yt96.inf))
yt96 = yt96[Not(1), :]
reg_ea = lm(@formula(inf_diff1 ~ unem), yt96)
X_ea = getMats(formula(reg_ea), yt96)[2]
resid_ea = residuals(reg_ea)

# DW tests:
DW_s = DurbinWatsonTest(X_s, resid_s)
DW_ea = DurbinWatsonTest(X_ea, resid_ea)
println("DW_s: \n$DW_s\n")
println("DW_ea: \n$DW_ea")
```

Script 12.5: `Example-12-5.jl`

```
using PyCall, WooldridgeDatasets, GLM, DataFrames
# install Python's statsmodels with: using Conda; Conda.add("statsmodels")
sm = pyimport("statsmodels.api")
include("../03/getMats.jl")

barium = DataFrame(wooldridge("barium"))

# definition of model and hypotheses:
f = @formula(log(chnimp) ~ 1 + log(chempi) + log(gas) + log(rtwex) +
                           befile6 + affile6 + afdec6)
xy = getMats(f, barium)
y = xy[1]
X = xy[2]

# perform the Cochrane-Orcutt estimation (iterative procedure):
reg = sm.GLSAR(y, X)
CORC_results = reg.iterative_fit(maxiter=100)

reg_rho = reg.rho
table = DataFrame(
    coefnames=["Intercept", "log(chempi)", "log(gas)", "log(rtwex)",
        "befile6", "affile6", "afdec6"],
    b_CORC=CORC_results.params,
    se_CORC=round.(CORC_results.bse, digits=5))

println("reg_rho = $reg_rho\n")
println("table: \n$table")
```

Script 12.6: `calc-hac-se.jl`

```
using LinearAlgebra

# for details, see Equations 12.41 - 12.43 in Wooldridge (2019)
function calc_hac_se(reg, g)
    n = nobs(reg)
    X = reg.mm.m
    n = size(X, 1)
    K = size(X, 2)
    u = residuals(reg)
    ser = sqrt(sum(u .^ 2) / (n - K))
    se_ols = coeftable(reg).cols[2]
    se_hac = zeros(K)

    for k in 1:K
        yk = X[:, k]
        Xk = X[:, (1:K).!=k]
        bk = inv(transpose(Xk) * Xk) * transpose(Xk) * yk
        rk = yk .- Xk * bk
        ak = rk .* u
        vk = sum(ak .^ 2)
        for h in 1:g
            sum_h = 2 * (1 - h / (g + 1)) * sum(ak[(h+1):n] .* ak[1:(n-h)])
            vk = vk + sum_h
        end

        se_hac[k] = (se_ols[k] / ser)^2 * sqrt(vk)
    end
```

```
    return se_hac
end
```

Script 12.7: Example-12-1.jl

```
using WooldridgeDatasets, GLM, DataFrames
include("calc-hac-se.jl")

prminwge = DataFrame(wooldridge("prminwge"))
prminwge.time = prminwge.year .- 1949

# OLS with regular SE:
reg = lm(@formula(log(prepop) ~ log(mincov) + log(prgnp) +
                                log(usgnp) + time), prminwge)

# OLS with HAC SE:
hac_se = calc_hac_se(reg, 2)

# print different SEs:
table = DataFrame(coefficients=coeftable(reg).rownms,
    b=round.(coef(reg), digits=5),
    se_default=round.(coeftable(reg).cols[2], digits=5),
    hac_se=round.(hac_se, digits=5))
println("table: \n$table")
```

Script 12.8: Example-12-9.jl

```
using WooldridgeDatasets, GLM, DataFrames

nyse = DataFrame(wooldridge("nyse"))
nyse.ret = nyse.return
nyse.ret_lag1 = lag(nyse.ret, 1)
nyse = nyse[Not(1, 2), :]

# linear regression of model:
reg = lm(@formula(ret ~ ret_lag1), nyse)

# squared residuals:
nyse.resid_sq = residuals(reg) .^ 2
nyse.resid_sq_lag1 = lag(nyse.resid_sq, 1)

# model for squared residuals:
ARCHreg = lm(@formula(resid_sq ~ resid_sq_lag1), nyse)
table_ARCHreg = coeftable(ARCHreg)
println("table_ARCHreg: \n$table_ARCHreg")
```

Script 12.9: Example-ARCH.jl

```
using DataFrames, GLM, Dates, MarketData

# download data for "AAPL" (= Apple) and define start and end:
ticker = "AAPL"
start_date = DateTime(2007, 12, 31)
end_date = DateTime(2017, 01, 01)

# import data as DataFrame:
AAPL_data = DataFrame(yahoo(ticker,
    YahooOpt(period1=start_date, period2=end_date)))

# calculate return as the difference of logged prices:
```

```
AAPL_data.ret = vcat(missing, diff(log.(AAPL_data.AdjClose)))
AAPL_data.ret_lag1 = lag(AAPL_data.ret, 1)
AAPL_data = AAPL_data[Not(1, 2), :]

# AR(1) model for returns:
reg = lm(@formula(ret ~ ret_lag1), AAPL_data)

# squared residuals:
AAPL_data.resid_sq = residuals(reg) .^ 2
AAPL_data.resid_sq_lag1 = lag(AAPL_data.resid_sq, 1)

# model for squared residuals:
ARCHreg = lm(@formula(resid_sq ~ resid_sq_lag1), AAPL_data)
table_ARCHreg = coeftable(ARCHreg)
println("table_ARCHreg: \n$table_ARCHreg")
```

13. Scripts Used in Chapter 13

Script 13.1: `Example-13-2.jl`

```
using WooldridgeDatasets, GLM, DataFrames

cps78_85 = DataFrame(wooldridge("cps78_85"))

# OLS results including interaction terms:
reg = lm(@formula(lwage ~ y85 * (educ + female) + exper +
                          ((exper^2) / 100) + union), cps78_85)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Script 13.2: `Example-13-3-1.jl`

```
using WooldridgeDatasets, GLM, DataFrames, RegressionTables

kielmc = DataFrame(wooldridge("kielmc"))
kielmc.is1981 = kielmc.year .== 1981

# separate regressions for 1978 and 1981:
y78 = subset(kielmc, :year => ByRow(==(1978)))
reg78 = lm(@formula(rprice ~ nearinc), y78)

y81 = subset(kielmc, :year => ByRow(==(1981)))
reg81 = lm(@formula(rprice ~ nearinc), y81)

# joint regression including an interaction term:
reg_joint = lm(@formula(rprice ~ nearinc * is1981), kielmc)

# print results with RegressionTables:
regtable(reg78, reg81, reg_joint)
```

Script 13.3: `Example-13-3-2.jl`

```
using WooldridgeDatasets, GLM, DataFrames

kielmc = DataFrame(wooldridge("kielmc"))
kielmc.is1981 = kielmc.year .== 1981

# difference in difference (DiD):
```

```
reg_did = lm(@formula(log(rprice) ~ nearinc * is1981), kielmc)
table_did = coeftable(reg_did)
println("table_did: \n$table_did\n")

# DiD with control variables:
reg_didC = lm(@formula(log(rprice) ~ nearinc * is1981 + age + (age^2) +
                                     log(intst) + log(land) + log(area) +
                                     rooms + baths), kielmc)
table_didC = coeftable(reg_didC)
println("table_didC: \n$table_didC")
```

Script 13.4: `Example-FD.jl`

```
using WooldridgeDatasets, GLM, DataFrames

crime2 = DataFrame(wooldridge("crime2"))

# create an index in this balanced data set by combining two vectors:
id_tmp = 1:46
crime2.id = sort(vcat(id_tmp, id_tmp))

# sort data by id and year:
sort!(crime2, [:id, :year])

# manually calculate first differences per entity for crmrte and unem:
grouped_df = groupby(crime2, :id)
diff_df = DataFrame(id=id_tmp)
diff_df.crmrte_diff1 = combine(grouped_df, :crmrte => diff).crmrte_diff
diff_df.unem_diff1 = combine(grouped_df, :unem => diff).unem_diff

preview = diff_df[1:5, :]
println("preview: \n$preview\n")

# estimate FD model with OLS on differenced data:
reg_sm = lm(@formula(crmrte_diff1 ~ unem_diff1), diff_df)
table_sm = coeftable(reg_sm)
println("table_sm: \n$table_sm")
```

Script 13.5: `Example-13-9.jl`

```
using WooldridgeDatasets, GLM, DataFrames

crime4 = DataFrame(wooldridge("crime4"))
crime4.lcrmrte = log.(crime4.crmrte)

# sort data by county and year:
sort!(crime4, [:county, :year])

# manually calculate first differences for multiple variables:
vars_to_diff = ["lcrmrte", "d83", "d84", "d85", "d86", "d87",
    "lprbarr", "lprbconv", "lprbpris", "lavgsen", "lpolpc"]
grouped_df = groupby(crime4, :county)
diff_df = DataFrame()

for i in vars_to_diff
    tmp_diff_i = combine(grouped_df, Symbol(i) => diff)[:, 2]
    diff_df[!, i] = tmp_diff_i
end

# estimate FD model:
```

```
reg = lm(@formula(lcrmrte ~ d83 + d84 + d85 + d86 + d87 +
                            lprbarr + lprbconv + lprbpris +
                            lavgsen + lpolpc), diff_df)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

14. Scripts Used in Chapter 14

Script 14.1: `Example-14-2.jl`

```
using WooldridgeDatasets, DataFrames, Econometrics

wagepan = DataFrame(wooldridge("wagepan"))

# FE model estimation:
reg = fit(EconometricModel,
    @formula(lwage ~ married + union +
                     d81 + d81 + d82 + d83 + d84 + d85 + d86 + d87 +
                     d81 & educ + d81 & educ + d82 & educ + d83 & educ +
                     d84 & educ + d85 & educ + d86 & educ + d87 & educ +
                     absorb(nr)),
    wagepan)
table_reg = coeftable(reg)
println("table_reg: \n$table_reg")
```

Script 14.2: `Example-14-4.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Econometrics

wagepan = DataFrame(wooldridge("wagepan"))

# estimate different models:
reg_ols = lm(@formula(lwage ~ educ + black + hisp + exper + (exper^2) +
                              married + union + year),
    wagepan,
    contrasts=Dict(:year => DummyCoding()))

reg_re = fit(RandomEffectsEstimator,
    @formula(lwage ~ educ + black + hisp + exper + (exper^2) +
                     married + union + year),
    wagepan,
    panel=:nr,
    time=:year,
    contrasts=Dict(:year => DummyCoding()))

reg_fe = fit(EconometricModel,
    @formula(lwage ~ (exper^2) + married + union + year + absorb(nr)),
    wagepan,
    contrasts=Dict(:year => DummyCoding()))

# print results:
table_ols = coeftable(reg_ols)
println("table_ols: \n$table_ols\n")

table_re = coeftable(reg_re)
println("table_re: \n$table_re\n")
```

```
table_fe = coeftable(reg_fe)
println("table_fe: \n$table_fe")
```

Script 14.3: **Example-Dummy-CRE.jl**

```
using WooldridgeDatasets, GLM, DataFrames, Econometrics, Statistics

wagepan = DataFrame(wooldridge("wagepan"))

# include group specific means:
grouped_means = combine(groupby(wagepan, :nr), [:married, :union] .=> mean)
wagepan = innerjoin(grouped_means, wagepan, on=:nr)

# estimate FE parameters in 3 different ways:
reg_we = fit(EconometricModel,
    @formula(lwage ~ married + union + absorb(nr) +
                     d81 + d81 + d82 + d83 + d84 + d85 + d86 + d87 +
                     d81 & educ + d81 & educ + d82 & educ + d83 & educ +
                     d84 & educ + d85 & educ + d86 & educ + d87 & educ),
    wagepan)

reg_dum = lm(@formula(lwage ~ married + union + year * educ + nr),
    wagepan,
    contrasts=Dict(:year => DummyCoding(), :nr => DummyCoding()))


reg_cre = fit(RandomEffectsEstimator,
    @formula(lwage ~ married + union + year * educ +
                     married_mean + union_mean),
    wagepan,
    panel=:nr,
    time=:year,
    contrasts=Dict(:year => DummyCoding()))

# compare to RE estimates:
reg_re = fit(RandomEffectsEstimator,
    @formula(lwage ~ married + union + year * educ),
    wagepan,
    panel=:nr,
    time=:year,
    contrasts=Dict(:year => DummyCoding()))

# print results for married and union:
table = DataFrame(coef_names=["married", "union"],
    b_we=round.(coef(reg_we)[[2, 3]], digits=5),
    b_dum=round.(coef(reg_dum)[[2, 3]], digits=5),
    b_cre=round.(coef(reg_cre)[[2, 3]], digits=5),
    b_re=round.(coef(reg_re)[[2, 3]], digits=5))
println("table:\n $table")
```

Script 14.4: **Example-CRE.jl**

```
using WooldridgeDatasets, GLM, DataFrames, Econometrics, Statistics

wagepan = DataFrame(wooldridge("wagepan"))

# include group specific means:
grouped_means = combine(groupby(wagepan, :nr), [:married, :union] .=> mean)
wagepan = innerjoin(grouped_means, wagepan, on=:nr)
```

```
# estimate CRE:
reg_CRE = fit(RandomEffectsEstimator,
    @formula(lwage ~ married + union + educ + black +
                     hisp + married_mean + union_mean),
    wagepan,
    panel=:nr,
    time=:year)
table_reg = coeftable(reg_CRE)
println("table_reg: \n$table_reg")
```

15. Scripts Used in Chapter 15

Script 15.1: `Example-15-1.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Econometrics, Statistics

mroz_wm = DataFrame(wooldridge("mroz"))

# restrict to non-missing wage observations:
mroz = mroz_wm[.!ismissing.(mroz_wm.wage), :]

# OLS slope parameter manually:
cov_yz = cov(mroz.lwage, mroz.fatheduc)
cov_xy = cov(mroz.educ, mroz.lwage)
cov_xz = cov(mroz.educ, mroz.fatheduc)
var_x = var(mroz.educ)
x_bar = mean(mroz.educ)
y_bar = mean(mroz.lwage)
b_ols_man = cov_xy / var_x
println("b_ols_man = $b_ols_man\n")

# IV slope parameter manually:
b_iv_man = cov_yz / cov_xz
println("b_iv_man = $b_iv_man\n")

# OLS automatically:
reg_ols = lm(@formula(lwage ~ educ), mroz)
table_ols = coeftable(reg_ols)
println("table_ols: \n$table_ols\n")

# IV automatically:
reg_iv = fit(EconometricModel,
    @formula(lwage ~ (educ ~ fatheduc)), mroz)
table_iv = coeftable(reg_iv)
println("table_iv: \n$table_iv")
```

Script 15.2: `Example-15-4.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Econometrics

card = DataFrame(wooldridge("card"))

# checking for relevance with reduced form:
reg_redf = lm(@formula(educ ~ nearc4 + exper + (exper^2) + black +
                              smsa + south + smsa66 + reg662 +
                              reg663 + reg664 + reg665 + reg666 +
                              reg667 + reg668 + reg669), card)
table_redf = coeftable(reg_redf)
```

```
println("table_redf: \n$table_redf\n")

# OLS:
reg_ols = lm(@formula(log(wage) ~ educ + exper + (exper^2) + black +
                                  smsa + south + smsa66 + reg662 +
                                  reg663 + reg664 + reg665 + reg666 +
                                  reg667 + reg668 + reg669), card)
table_ols = coeftable(reg_ols)
println("table_ols: \n$table_ols\n")


# IV automatically:
reg_iv = fit(EconometricModel,
    @formula(log(wage) ~ exper + (exper^2) + black + smsa +
                         south + smsa66 + reg662 + reg663 +
                         reg664 + reg665 + reg666 + reg667 +
                         reg668 + reg669 + (educ ~ nearc4)), card)
table_iv = coeftable(reg_iv)
println("table_iv: \n$table_iv")
```

Script 15.3: `Example-15-5.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Econometrics

mroz_wm = DataFrame(wooldridge("mroz"))

# restrict to non-missing wage observations:
mroz = mroz_wm[.!ismissing.(mroz_wm.wage), :]

# 1st stage (reduced form):
reg_redf = lm(@formula(educ ~ exper + (exper^2) +
                              motheduc + fatheduc), mroz)
mroz.educ_fitted = predict(reg_redf)
table_redf = coeftable(reg_redf)
println("table_redf: \n$table_redf\n")

# 2nd stage:
reg_secstg = lm(@formula(log(wage) ~ educ_fitted + exper +
                                     (exper^2)), mroz)
table_reg_secstg = coeftable(reg_secstg)
println("table_reg_secstg: \n$table_reg_secstg\n")

# IV automatically:
reg_iv = fit(EconometricModel,
    @formula(log(wage) ~ exper + (exper^2) +
                         (educ ~ motheduc + fatheduc)), mroz)
table_iv = coeftable(reg_iv)
println("table_iv: \n$table_iv")
```

Script 15.4: `Example-15-7.jl`

```
using WooldridgeDatasets, GLM, DataFrames

mroz_wm = DataFrame(wooldridge("mroz"))

# restrict to non-missing wage observations:
mroz = mroz_wm[.!ismissing.(mroz_wm.wage), :]

# 1st stage (reduced form):
reg_redf = lm(@formula(educ ~ exper + (exper^2) +
```

```
                              motheduc + fatheduc), mroz)
mroz.resid = residuals(reg_redf)

# 2nd stage:
reg_secstg = lm(@formula(log(wage) ~ resid + educ +
                                     exper + (exper^2)), mroz)
table_reg_secstg = coeftable(reg_secstg)
println("table_reg_secstg: \n$table_reg_secstg")
```

Script 15.5: `Example-15-8.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Econometrics, Distributions

mroz_wm = DataFrame(wooldridge("mroz"))

# restrict to non-missing wage observations:
mroz = mroz_wm[.!ismissing.(mroz_wm.wage), :]

# IV regression:
reg_iv = fit(EconometricModel,
    @formula(log(wage) ~ exper + (exper^2) +
                         (educ ~ motheduc + fatheduc)), mroz)
table_iv = coeftable(reg_iv)
println("table_iv: \n$table_iv\n")

# auxiliary regression:
mroz.resid_iv = residuals(reg_iv)
reg_aux = lm(@formula(resid_iv ~ exper + (exper^2) +
                                 motheduc + fatheduc), mroz)

# calculations for test:
R2 = r2(reg_aux)
n = nobs(reg_aux)
teststat = n * R2
pval = 1 - cdf(Chisq(1), teststat)

println("R2 = $R2\n")
println("n = $n\n")
println("teststat = $teststat\n")
println("pval = $pval")
```

Script 15.6: `Example-15-10.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Econometrics

jtrain = DataFrame(wooldridge("jtrain"))

# define panel data (for 1987 and 1988 only) and sort:
jtrain_8788 = subset(jtrain, :year => ByRow(<=(1988)))
sort!(jtrain_8788, [:fcode, :year])

# manual computation of deviations of entity means:
grouped_df = groupby(jtrain_8788, :fcode)
diff_df = DataFrame(fcode=unique(jtrain_8788.fcode))
diff_df.lscrap_diff1 = combine(grouped_df, :lscrap => diff).lscrap_diff
diff_df.hrsemp_diff1 = combine(grouped_df, :hrsemp => diff).hrsemp_diff
diff_df.grant_diff1 = combine(grouped_df, :grant => diff).grant_diff

# IV regression:
reg_iv = fit(EconometricModel,
```

```
    @formula(lscrap_diff1 ~ (hrsemp_diff1 ~ grant_diff1)), diff_df)
table_iv = coeftable(reg_iv)
println("table_iv: \n$table_iv")
```

16. Scripts Used in Chapter 16

Script 16.1: `Example-16-5-2SLS-1.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Econometrics, Statistics

mroz_wm = DataFrame(wooldridge("mroz"))

# restrict to non-missing wage observations:
mroz = mroz_wm[.!ismissing.(mroz_wm.wage), :]

# 2SLS regressions:
reg_iv1 = fit(EconometricModel,
    @formula(hours ~ educ + age + kidslt6 + nwifeinc +
                     (log(wage) ~ exper + (exper^2))), mroz)
table_iv1 = coeftable(reg_iv1)
println("table_iv1: \n$table_iv1\n")

reg_iv2 = fit(EconometricModel,
    @formula(log(wage) ~ educ + exper + (exper^2) +
                         (hours ~ age + kidslt6 + nwifeinc)), mroz)
table_iv2 = coeftable(reg_iv2)
println("table_iv2: \n$table_iv2\n")

cor_u1u2 = cor(residuals(reg_iv1), residuals(reg_iv2))
println("cor_u1u2 =$cor_u1u2")
```

Script 16.2: `Example-16-5-2SLS-2.jl`

```
using WooldridgeDatasets, GLM, DataFrames, PyCall
include("../03/getMats.jl")

# install Python's linearmodels with: using Conda; Conda.add("linearmodels")
iv = pyimport("linearmodels.iv")

mroz_wm = DataFrame(wooldridge("mroz"))

# restrict to non-missing wage observations:
mroz = mroz_wm[.!ismissing.(mroz_wm.wage), :]

# prepare for equation 1:
f1 = @formula(hours ~ 1 + educ + age + kidslt6 + nwifeinc)
yexog = getMats(f1, mroz)
y_eq1 = yexog[1]
exog_mat_eq1 = yexog[2]

f2 = @formula(1 ~ 0 + log(wage))
endo_mat_eq1 = getMats(f2, mroz)[2]

f3 = @formula(1 ~ 0 + exper + (exper^2))
iv_mat_eq1 = getMats(f3, mroz)[2]

# prepare for equation 2:
f1 = @formula(log(wage) ~ 1 + educ + exper + (exper^2))
```

```
yexog = getMats(f1, mroz)
y_eq2 = yexog[1]
exog_mat_eq2 = yexog[2]

f2 = @formula(1 ~ 0 + hours)
endo_mat_eq2 = getMats(f2, mroz)[2]

f3 = @formula(1 ~ 0 + age + kidslt6 + nwifeinc)
iv_mat_eq2 = getMats(f3, mroz)[2]

# use Python's linearmodels:
reg_iv1 = iv.IV2SLS(y_eq1, exog_mat_eq1, endo_mat_eq1, iv_mat_eq1)
results_iv1 = reg_iv1.fit(cov_type="unadjusted", debiased=true)
println("results_iv1: \n$results_iv1\n")

reg_iv2 = iv.IV2SLS(y_eq2, exog_mat_eq2, endo_mat_eq2, iv_mat_eq2)
results_iv2 = reg_iv2.fit(cov_type="unadjusted", debiased=true)
println("results_iv2: \n$results_iv2")
```

Script 16.3: `Example-16-5-3SLS.jl`

```
using WooldridgeDatasets, GLM, DataFrames, PyCall
include("../03/getMats.jl")
iv3 = pyimport("linearmodels.system")

mroz_wm = DataFrame(wooldridge("mroz"))

# restrict to non-missing wage observations:
mroz = mroz_wm[.!ismissing.(mroz_wm.wage), :]

# prepare for equation 1:
f1 = @formula(hours ~ 1 + educ + age + kidslt6 + nwifeinc)
yexog = getMats(f1, mroz)
y_eq1 = yexog[1]
exog_mat_eq1 = yexog[2]

f2 = @formula(1 ~ 0 + log(wage))
endo_mat_eq1 = getMats(f2, mroz)[2]

f3 = @formula(1 ~ 0 + exper + (exper^2))
iv_mat_eq1 = getMats(f3, mroz)[2]

# prepare for equation 2:
f1 = @formula(log(wage) ~ 1 + educ + exper + (exper^2))
yexog = getMats(f1, mroz)
y_eq2 = yexog[1]
exog_mat_eq2 = yexog[2]

f2 = @formula(1 ~ 0 + hours)
endo_mat_eq2 = getMats(f2, mroz)[2]

f3 = @formula(1 ~ 0 + age + kidslt6 + nwifeinc)
iv_mat_eq2 = getMats(f3, mroz)[2]

# use Python's linearmodels:
reg_3sls = iv3.IV3SLS(Dict([
    ("eq1", (y_eq1, exog_mat_eq1, endo_mat_eq1, iv_mat_eq1)),
    ("eq2", (y_eq2, exog_mat_eq2, endo_mat_eq2, iv_mat_eq2))]))
```

```
results_3sls = reg_3sls.fit(cov_type="unadjusted", debiased=true)
println("results_3sls: \n$results_3sls")
```

17. Scripts Used in Chapter 17

Script 17.1: Example-17-1-1.jl

```
using WooldridgeDatasets, GLM, DataFrames
include("../08/calc-white-se.jl")

mroz = DataFrame(wooldridge("mroz"))

# estimate linear probability model:
reg_lin = lm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) +
                             age + kidslt6 + kidsge6), mroz)
hc0 = calc_white_se(reg_lin, mroz)

table_reg_lin = DataFrame(
    coefficients=coeftable(reg_lin).rownms,
    b=round.(coef(reg_lin), digits=5),
    se_white=hc0)
println("table_reg_lin: \n$table_reg_lin")
```

Script 17.2: Example-17-1-2.jl

```
using WooldridgeDatasets, GLM, DataFrames

mroz = DataFrame(wooldridge("mroz"))

# estimate linear probability model:
reg_lin = lm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) +
                             age + kidslt6 + kidsge6), mroz)

# predictions for two "extreme" women:
X_new = DataFrame(nwifeinc=[100, 0], educ=[5, 17],
     exper=[0, 30], age=[20, 52],
     kidslt6=[2, 0], kidsge6=[0, 0])
predictions = round.(predict(reg_lin, X_new), digits=5)

print("predictions = $predictions")
```

Script 17.3: Example-17-1-3.jl

```
using WooldridgeDatasets, GLM, DataFrames

mroz = DataFrame(wooldridge("mroz"))

# estimate logit model:
reg_logit = glm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) + age +
                                kidslt6 + kidsge6),
    mroz, Binomial(), LogitLink())
table_reg_logit = coeftable(reg_logit)
println("table_reg_logit: \n$table_reg_logit\n")

# log likelihood value:
ll = deviance(reg_logit) / -2
println("ll = $ll\n")
```

```
# McFadden's pseudo R2:
reg_logit_null = glm(@formula(inlf ~ 1), mroz, Binomial(), LogitLink())
ll_null = deviance(reg_logit_null) / -2
pr2 = 1 - ll / ll_null
println("pr2 = $pr2")
```

Script 17.4: `Example-17-1-4.jl`

```
using WooldridgeDatasets, GLM, DataFrames

mroz = DataFrame(wooldridge("mroz"))

# estimate probit model:
reg_probit = glm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) + age +
                                 kidslt6 + kidsge6),
    mroz, Binomial(), ProbitLink())
table_reg_probit = coeftable(reg_probit)
println("table_reg_probit: \n$table_reg_probit\n")

# log likelihood value:
ll = deviance(reg_probit) / -2
println("ll=$ll\n")

# McFadden's pseudo R2:
reg_probit_null = glm(@formula(inlf ~ 1), mroz, Binomial(), ProbitLink())
ll_null = deviance(reg_probit_null) / -2
pr2 = 1 - ll / ll_null
println("pr2 = $pr2")
```

Script 17.5: `Example-17-1-5.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Distributions

mroz = DataFrame(wooldridge("mroz"))

# estimate probit model:
reg_probit = glm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) + age +
                                 kidslt6 + kidsge6),
    mroz, Binomial(), ProbitLink())
ll = deviance(reg_probit) / -2

# test of overall significance (test statistic and p value):
reg_probit_null = glm(@formula(inlf ~ 1), mroz, Binomial(), ProbitLink())
ll_null = deviance(reg_probit_null) / -2
lr1 = 2 * (ll - ll_null)
pval_all = 1 - cdf(Chisq(7), lr1)
println("lr1 = $lr1\n")
println("pval_all = $pval_all\n")

# likelihood ratio statistic test of H0 (experience and age are irrelevant):
reg_probit_hyp = glm(@formula(inlf ~ nwifeinc + educ + kidslt6 + kidsge6),
    mroz, Binomial(), ProbitLink())
ll_hyp = deviance(reg_probit_hyp) / -2
lr2 = 2 * (ll - ll_hyp)
pval_hyp = 1 - cdf(Chisq(3), lr2)
println("lr2 = $lr2\n")
println("pval_hyp = $pval_hyp")
```

Script 17.6: `Example-17-1-6.jl`

```
using WooldridgeDatasets, GLM, DataFrames

mroz = DataFrame(wooldridge("mroz"))

# estimate models:
reg_lin = lm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) + age +
                             kidslt6 + kidsge6), mroz)
reg_logit = glm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) + age +
                                kidslt6 + kidsge6),
     mroz, Binomial(), LogitLink())
reg_probit = glm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) + age +
                                 kidslt6 + kidsge6),
     mroz, Binomial(), ProbitLink())

# predictions for two "extreme" women:
X_new = DataFrame(nwifeinc=[100, 0], educ=[5, 17],
     exper=[0, 30], age=[20, 52],
     kidslt6=[2, 0], kidsge6=[0, 0])
predictions_lin = round.(predict(reg_lin, X_new), digits=5)
predictions_logit = round.(predict(reg_logit, X_new), digits=5)
predictions_probit = round.(predict(reg_probit, X_new), digits=5)

println("predictions_lin = $predictions_lin\n")
println("predictions_logit = $predictions_logit\n")
println("predictions_probit = $predictions_probit")
```

Script 17.7: `Binary-Predictions.jl`

```
using Distributions, GLM, Random, Plots, DataFrames

# set the random seed:
Random.seed!(12345)

y = rand(Binomial(1, 0.5), 100)
x = rand(Normal(), 100) + 2 * y
sim_data = DataFrame(y=y, x=x)

# estimation:
reg_lin = lm(@formula(y ~ x), sim_data)
reg_logit = glm(@formula(y ~ x), sim_data, Binomial(), LogitLink())
reg_probit = glm(@formula(y ~ x), sim_data, Binomial(), ProbitLink())

# prediction for regular grid of x values:
X_new = DataFrame(x=range(minimum(x), maximum(x), length=50))
predictions_lin = predict(reg_lin, X_new)
predictions_logit = predict(reg_logit, X_new)
predictions_probit = predict(reg_probit, X_new)

# scatter plot and fitted values:
scatter(x, y, label=false, color="black", legend=:topleft)
plot!(X_new.x, predictions_lin, linewidth=2,
    label="linear", color="black")
plot!(X_new.x, predictions_logit, linewidth=2,
    label="logit", color="black", linestyle=:dash)
plot!(X_new.x, predictions_probit, linewidth=2,
    label="probit", color="black", linestyle=:dot)
ylabel!("y")
xlabel!("x")
savefig("JlGraphs/Binary-Predictions.pdf")
```

Script 17.8: `Binary-Margeff.jl`

```
using Distributions, GLM, Random, Plots, DataFrames

# set the random seed:
Random.seed!(12345)

y = rand(Binomial(1, 0.5), 100)
x = rand(Normal(), 100) + 2 * y
sim_data = DataFrame(y=y, x=x)

# estimation:
reg_lin = lm(@formula(y ~ x), sim_data)
reg_logit = glm(@formula(y ~ x), sim_data, Binomial(), LogitLink())
reg_probit = glm(@formula(y ~ x), sim_data, Binomial(), ProbitLink())

# partial effects:
PE_lin = range(coef(reg_lin)[2], coef(reg_lin)[2], length=100)

coefs_logit = coeftable(reg_logit).cols[1]
xb_logit = reg_logit.mm.m * coefs_logit
factor_logit = pdf.(Logistic(), xb_logit)
PE_logit = coefs_logit[2] * factor_logit

coefs_probit = coeftable(reg_probit).cols[1]
xb_probit = reg_probit.mm.m * coefs_probit
factor_probit = pdf.(Normal(), xb_probit)
PE_probit = coefs_probit[2] * factor_probit

# plot PE's:
scatter(x, PE_lin, markershape=:circle, label="linear",
    color="black", legend=:topleft)
scatter!(x, PE_logit, markershape=:cross, label="logit",
    color="black", legend=:topleft)
scatter!(x, PE_probit, markershape=:star, label="probit",
    color="black", legend=:topleft)
ylabel!("partial effects")
xlabel!("x")
savefig("JlGraphs/Binary-margeff.pdf")
```

Script 17.9: `Example-17-1-7.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Statistics,
    Distributions, LinearAlgebra

mroz = DataFrame(wooldridge("mroz"))

# estimate models:
reg_lin = lm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) + age +
                             kidslt6 + kidsge6), mroz)
reg_logit = glm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) + age +
                                kidslt6 + kidsge6),
    mroz, Binomial(), LogitLink())
reg_probit = glm(@formula(inlf ~ nwifeinc + educ + exper + (exper^2) + age +
                                 kidslt6 + kidsge6),
    mroz, Binomial(), ProbitLink())
```

```
# average partial effects:
APE_lin = coef(reg_lin)

coefs_logit = coef(reg_logit)
xb_logit = reg_logit.mm.m * coefs_logit
factor_logit = mean(pdf.(Logistic(), xb_logit))
APE_logit = coefs_logit * factor_logit

coefs_probit = coef(reg_probit)
xb_probit = reg_probit.mm.m * coefs_probit
factor_probit = mean(pdf.(Normal(), xb_probit))
APE_probit = coefs_probit * factor_probit

# print results:
table_manual = DataFrame(
    coef_names=coeftable(reg_lin).rownms,
    APE_lin=round.(APE_lin, digits=5),
    APE_logit=round.(APE_logit, digits=5),
    APE_probit=round.(APE_probit, digits=5))
println("table_manual: \n$table_manual")
```

Script 17.10: `Example-17-3.jl`

```
using WooldridgeDatasets, GLM, DataFrames

crime1 = DataFrame(wooldridge("crime1"))

# estimate linear model:
reg_lin = lm(@formula(narr86 ~ pcnv + avgsen + tottime + ptime86 + qemp86 +
                               inc86 + black + hispan + born60), crime1)
table_lin = coeftable(reg_lin)
println("table_lin: \n$table_lin\n")

# estimate Poisson model:
reg_poisson = glm(@formula(narr86 ~ pcnv + avgsen + tottime + ptime86 + qemp86 +
                                    inc86 + black + hispan + born60),
    crime1, Poisson())
table_poisson = coeftable(reg_poisson)
println("table_poisson: \n$table_poisson\n")

# estimate Quasi-Poisson model:
yhat = predict(reg_poisson)
resid = crime1.narr86 .- yhat
sigma_sq = 1 / (2725 - 9 - 1) * sum(resid .^ 2 ./ yhat)
table_qpoisson = coeftable(reg_poisson)
table_qpoisson.cols[2] = table_qpoisson.cols[2] * sqrt(sigma_sq)
println("table_qpoisson: \n$table_qpoisson")
```

Script 17.11: `Tobit-CondMean.jl`

```
using Random, Distributions, Statistics, Plots

# set the random seed:
Random.seed!(12345)

x = sort(rand(Normal(), 100) .+ 4)
xb = -4 .+ 1 * x
y_star = xb .+ rand(Normal(), 100)
y = copy(y_star)
y[y_star.<0] .= 0
```

```
# conditional means:
Eystar = xb
Ey = cdf.(Normal(), xb) .* xb .+ pdf.(Normal(), xb)

# plot data and conditional means:
hline([0], color="grey", label=false, legend=:topleft)
scatter!(x, y_star, color="black", markershape=:xcross, label="y*")
scatter!(x, y, color="black", markershape=:cross, label="y")
plot!(x, Eystar, color="black", linewidth=2, linestyle=:solid, label="E(y*)")
plot!(x, Ey, color="black", linewidth=2, linestyle=:dash, label="E(y)")
ylabel!("y")
xlabel!("x")
savefig("JlGraphs/Tobit-CondMean.pdf")
```

Script 17.12: `Example-17-2.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Statistics, Distributions,
    LinearAlgebra, Optim
include("../03/getMats.jl")

# load data and build data matrices:
mroz = DataFrame(wooldridge("mroz"))
f = @formula(hours ~ 1 + nwifeinc + educ + exper +
                     (exper^2) + age + kidslt6 + kidsge6)
xy = getMats(f, mroz)
y = xy[1]
X = xy[2]

# define a function that returns the negative log likelihood per observation
# (for details on the implementation see Wooldridge (2019), formula 17.22):
function ll_tobit(params, y, X)
    p = size(X, 2)
    beta = params[1:p]
    sigma = exp(params[p+1])
    y_hat = X * beta
    y_eq = (y .== 0)
    y_g = (y .> 0)
    ll = zeros(length(y))
    ll[y_eq] = log.(cdf.(Normal(), -y_hat[y_eq] / sigma))
    ll[y_g] = log.(pdf.(Normal(), (y.-y_hat)[y_g] / sigma)) .- log(sigma)
    # return the negative sum of log likelihoods for each observation:
    return -sum(ll)
end


# generate starting solution:
reg_ols = lm(@formula(hours ~ nwifeinc + educ + exper + (exper^2) +
                              age + kidslt6 + kidsge6), mroz)
resid_ols = residuals(reg_ols)
sigma_start = log(sum(resid_ols .^ 2) / length(resid_ols))
params_start = vcat(coef(reg_ols), sigma_start)

# maximize the log likelihood = minimize the negative of the log likelihood:
optimum = optimize(par -> ll_tobit(par, y, X), params_start, Newton())
mle_est = Optim.minimizer(optimum)
ll = -optimum.minimum

# print results:
table_mle = DataFrame(
```

```
    coef_names=vcat(coeftable(reg_ols).rownms, "exp_sigma"),
    mle_est=round.(mle_est, digits=5))
println("table_mle: \n$table_mle\n")
println("ll = $ll")
```

Script 17.13: `Example-17-4.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Statistics, Distributions,
        LinearAlgebra, Optim

# load data and build data matrices:
recid = DataFrame(wooldridge("recid"))
f = @formula(ldurat ~ 1 + workprg + priors + tserved +
                      felon + alcohol + drugs + black +
                      married + educ + age)
xy = getMats(f, recid)
y = xy[1]
X = xy[2]

# define dummy for censored observations:
censored = recid.cens .!= 0

# generate starting solution:
reg_ols = lm(@formula(ldurat ~ workprg + priors + tserved +
                               felon + alcohol + drugs +
                               black + married +
                               educ + age), recid)
resid_ols = residuals(reg_ols)
sigma_start = log(sum(resid_ols .^ 2) / length(resid_ols))
params_start = vcat(coef(reg_ols), sigma_start)

# define a function that returns the negative log likelihood per observation:
function ll_censreg(params, y, X, cens)
        p = size(X, 2)
        beta = params[1:p]
        sigma = exp(params[p+1])
        y_hat = X * beta
        ll = zeros(length(y))
        # uncensored:
        ll[.!cens] = log.(pdf.(Normal(),
                (y.-y_hat)[.!cens] / sigma)) .- log(sigma)
        # censored:
        ll[cens] = log.(cdf.(Normal(), -(y.-y_hat)[cens] / sigma))

        # return the negative sum of log likelihoods for each observation:
        return -sum(ll)
end

# maximize the log likelihood = minimize the negative of the log likelihood:
optimum = optimize(par -> ll_censreg(par, y, X, censored), params_start, Newton())
mle_est = Optim.minimizer(optimum)
ll = -optimum.minimum

# print results of MLE:
table_mle = DataFrame(
        coef_names=vcat(coeftable(reg_ols).rownms, "exp_sigma"),
        mle_est=round.(mle_est, digits=5))
println("table_mle: \n$table_mle\n")
println("ll = $ll")
```

Script 17.14: `TruncReg-Simulation.jl`

```
using GLM, Random, Distributions, Statistics, Plots, DataFrames

# set the random seed:
Random.seed!(12345)

x = sort(rand(Normal(), 100) .+ 4)
y = -4 .+ 1 * x .+ rand(Normal(), 100)
compl = DataFrame(x=x, y=y)

sample = (y .> 0)

# complete observations and observed sample:
x_sample = x[sample]
y_sample = y[sample]

sample = DataFrame(x=x_sample, y=y_sample)

# predictions OLS:
reg_ols = lm(@formula(y ~ x), sample)
yhat_ols = fitted(reg_ols)

# predictions truncated regression:
reg_tr = lm(@formula(y ~ x), compl)
yhat_tr = fitted(reg_tr)


# plot data and conditional means:
hline([0], color="grey", label=false, legend=:topleft)
scatter!(compl.x, compl.y, color="black", markershape=:circle,
    markercolor=:white, label="all data")
scatter!(sample.x, sample.y, color="black", markershape=:circle,
    label="sample data")
plot!(sample.x, yhat_ols, color="black", linewidth=2,
    linestyle=:dash, label="OLS fit")
plot!(compl.x, yhat_tr, color="black", linewidth=2,
    linestyle=:solid, label="Trunc. Reg. fit")
ylabel!("y")
xlabel!("x")
savefig("JlGraphs/TruncReg-Simulation.pdf")
```

Script 17.15: `Example-17-5.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Distributions

# load data and build data matrices:
mroz = DataFrame(wooldridge("mroz"))

# step 1 (use all n observations to estimate a probit model of s_i on z_i):
reg_probit = glm(@formula(inlf ~ educ + exper +
                                 (exper^2) + nwifeinc +
                                 age + kidslt6 + kidsge6),
    mroz, Binomial(), ProbitLink())
pred_inlf_linpart = quantile.(Normal(), fitted(reg_probit))
mroz.inv_mills = pdf.(Normal(), pred_inlf_linpart) ./
                 cdf.(Normal(), pred_inlf_linpart)

# step 2 (regress y_i on x_i and inv_mills in sample selection):
mroz_subset = subset(mroz, :inlf => ByRow(==(1)))
reg_heckit = lm(@formula(lwage ~ educ + exper + (exper^2) +
```

```
                                    inv_mills), mroz_subset)

# print results:
table_reg_heckit = coeftable(reg_heckit)
println("table_reg_heckit: \n$table_reg_heckit")
```

18. Scripts Used in Chapter 18

Script 18.1: `Example-18-1.jl`

```
using WooldridgeDatasets, GLM, DataFrames

hseinv = DataFrame(wooldridge("hseinv"))

# add lags and detrend:
reg_trend = lm(@formula(linvpc ~ t), hseinv)
hseinv.linvpc_det = residuals(reg_trend)
hseinv.gprice_lag1 = lag(hseinv.gprice, 1)
hseinv.linvpc_det_lag1 = lag(hseinv.linvpc_det, 1)

# Koyck geometric d.l.:
reg_koyck = lm(@formula(linvpc_det ~ gprice +
                                     linvpc_det_lag1), hseinv)
table_koyck = coeftable(reg_koyck)
println("table_koyck: \n$table_koyck\n")

# rational d.l.:
reg_rational = lm(@formula(linvpc_det ~ gprice + linvpc_det_lag1 +
                                        gprice_lag1), hseinv)
table_rational = coeftable(reg_rational)
println("table_rational: \n$table_rational\n")

# calculate LRP as...
# gprice / (1 - linvpc_det_lag1):
lrp_koyck = coef(reg_koyck)[2] / (1 - coef(reg_koyck)[3])
println("lrp_koyck = $lrp_koyck\n")

# and (gprice + gprice_lag1) / (1 - linvpc_det_lag1):
lrp_rational = (coef(reg_rational)[2] + coef(reg_rational)[4]) /
               (1 - coef(reg_rational)[3])
println("lrp_rational = $lrp_rational")
```

Script 18.2: `Example-18-4.jl`

```
using WooldridgeDatasets, DataFrames, HypothesisTests

inven = DataFrame(wooldridge("inven"))
inven.lgdp = log.(inven.gdp)

# automated ADF:
adf_lag = 1
res_ADF_aut = ADFTest(inven.lgdp, :trend, adf_lag)
println("res_ADF_aut: \n$res_ADF_aut")
```

Script 18.3: `Simulate-Spurious-Regression-1.jl`

```
using Random, Distributions, Statistics, Plots, GLM, DataFrames
```

```
# set the random seed:
Random.seed!(12345)

# i.i.d. N(0,1) innovations:
n = 51
e = rand(Normal(), n)
e[1] = 0
a = rand(Normal(), n)
a[1] = 0

# independent random walks:
x = cumsum(a)
y = cumsum(e)
sim_data = DataFrame(y=y, x=x)

# regression:
reg = lm(@formula(y ~ x), sim_data)
reg_table = coeftable(reg)
println("reg_table: \n$reg_table")

# graph:
plot(x, color="black", linewidth=2, linestyle=:solid, label="x")
plot!(y, color="black", linewidth=2, linestyle=:dash, label="y")
ylabel!("y")
xlabel!("x")
savefig("JlGraphs/Simulate-Spurious-Regression-1.pdf")
```

Script 18.4: `Simulate-Spurious-Regression-2.jl`

```
using Random, Distributions, Statistics, Plots, GLM, DataFrames

# set the random seed:
Random.seed!(12345)

pvals = zeros(10000)

for i in 1:10000
    # i.i.d. N(0,1) innovations:
    n = 51
    e = rand(Normal(), n)
    e[1] = 0
    a = rand(Normal(), n)
    a[1] = 0

    # independent random walks:
    x = cumsum(a)
    y = cumsum(e)
    sim_data = DataFrame(y=y, x=x)

    # regression:
    reg = lm(@formula(y ~ x), sim_data)
    reg_table = coeftable(reg)

    # save the p value of x:
    pvals[i] = reg_table.cols[4][2]
end

# how often is p<=5%:
count_pval_smaller = sum(pvals .<= 0.05)  # counts true elements
println("count_pval_smaller = $count_pval_smaller\n")
```

```
# how often is p>5%:
count_pval_greater = sum(pvals .> 0.05)  # counts true elements
println("count_pval_greater = $count_pval_greater")
```

Script 18.5: `Example-18-8.jl`

```
using WooldridgeDatasets, GLM, DataFrames, Statistics, Plots

phillips = DataFrame(wooldridge("phillips"))

# estimate models:
yt96 = subset(phillips, :year => ByRow(<=(1996)))
reg_1 = lm(@formula(unem ~ unem_1), yt96)
reg_2 = lm(@formula(unem ~ unem_1 + inf_1), yt96)

# predictions for 1997-2003:
yf97 = subset(phillips, :year => ByRow(>(1996)))
pred_1 = round.(predict(reg_1, yf97), digits=5)
println("pred_1 = $pred_1\n")
pred_2 = round.(predict(reg_2, yf97), digits=5)
println("pred_2 = $pred_2\n")

# forecast errors:
e1 = yf97.unem .- pred_1
e2 = yf97.unem .- pred_2

# RMSE and MAE:
rmse1 = sqrt(mean(e1 .^ 2))
println("rmse1 = $rmse1\n")

rmse2 = sqrt(mean(e2 .^ 2))
println("rmse2 = $rmse2\n")

mae1 = mean(abs.(e1))
println("mae1 = $mae1\n")

mae2 = mean(abs.(e2))
println("mae2 = $mae2")

# graph:
plot(yf97.year, yf97.unem, color="black", linewidth=2,
    linestyle=:solid, label="unem", legend=:topleft)
plot!(yf97.year, pred_1, color="black", linewidth=2,
    linestyle=:dash, label="forecast without inflation")
plot!(yf97.year, pred_2, color="black", linewidth=2,
    linestyle=:dashdot, label="forecast with inflation")
ylabel!("unemployment")
xlabel!("time")
savefig("JlGraphs/Example-18-8.pdf")
```

19. Scripts Used in Chapter 19

Script 19.1: `ultimate-calcs.jl`

```
########################################################################
# Project X:
# "The Ultimate Question of Life, the Universe, and Everything"
```

```
# Project Collaborators: Mr. H, Mr. B
#
# Julia Script "ultimate-calcs"
# by: F Heiss
# Date of this version: December 1, 2022
########################################################################
# load packages:
using Dates

# create a time stamp:
ts = now()

# print to logfile.txt (write=true resets the logfile before writing output)
# in the provided path (make sure that the folder structure
# you may provide already exists):
open("Jlout/19/logfile.txt", write=true) do io
    println(io, "This is a log file from: \n $ts\n")
end

# the first calculation using the square root function:
result1 = sqrt(1764)
# print to logfile.txt but with keeping the previous results (append=true):
open("Jlout/19/logfile.txt", append=true) do io
    println(io, "result1: $result1\n")
end

# the second calculation reverses the first one:
result2 = result1^2
# print to logfile.txt but with keeping the previous results (append=true):
open("Jlout/19/logfile.txt", append=true) do io
    println(io, "result2: $result2")
end
```

Script 19.2: `ultimate-calcs2.jl`

```
# load packages:
using Dates

# create a time stamp:
ts = now()

# print to logfile2.txt (write=true resets the logfile before writing output)
# in the provided path (make sure that the folder structure
# you may provide already exists):
open("Jlout/19/logfile2.txt", write=true) do io
    println(io, "This is a log file from: \n $ts\n")

    # the first calculation using the square root function:
    result1 = sqrt(1764)
    # print to logfile2.txt:
    println(io, "result1: $result1\n")

    # the second calculation reverses the first one:
    result2 = result1^2
    # print to logfile2.txt:
    println(io, "result2: $result2")
end
```

Bibliography

Bates, D. and other contributors (2012): *GLM.jl Manual: Linear and generalized linear models in Julia*, https://juliastats.org/GLM.jl/stable/.

Bates, D., J. M. White, J. Bezanson, S. Karpinski, V. B. Shah, and other contributors (2012): *Distributions.jl*, https://juliastats.org/Distributions.jl/stable/.

Bhardwaj, A. (2020): *WooldridgeDatasets.jl*, https://docs.juliahub.com/WooldridgeDatasets/.

Boehm, J. (2017): *RegressionTables.jl*, https://github.com/jmboehm/RegressionTables.jl.

Bouchet-Valat, M. (2014): *FreqTables.jl*, https://github.com/nalimilan/FreqTables.jl.

Breloff, T. (2015): *Plots.jl*, https://docs.juliaplots.org/stable/.

——— (2016): *StatsPlots.jl*, https://docs.juliaplots.org/latest/generated/statsplots/.

Byrne, S. (2014): *KernelDensity.jl*, https://docs.juliahub.com/KernelDensity/.

Calderón, S. and J. Bayoán (2020): *Econometrics.jl*, vol. 1, The Open Journal.

Harris, H., EPRI, C. DuBois, J. M. White, M. Bouchet-Valat, B. Kamiński, and other contributors (2022): *DataFrames Documentation*, https://dataframes.juliadata.org/stable/.

Heiss, F. (2020): *Using R for Introductory Econometrics*, CreateSpace Independent Publishing Platform, 2 ed.

Heiss, F. and D. Brunner (2020): *Using Python for Introductory Econometrics*, CreateSpace Independent Publishing Platform, 2 ed.

Johnson, S. G. (2015): *PyCall.jl*, https://docs.juliahub.com/PyCall/.

Kleinschmidt, D. (2016): *StatsModels.jl Documentation*, https://juliastats.org/StatsModels.jl/.

Kluyver, T., B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and J. development team (2016): "Jupyter Notebooks - a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, ed. by F. Loizides and B. Schmidt, IOS Press, 87–90.

Kornblith, S. and other contributors (2012): *HypothesisTests.jl*, https://juliastats.org/HypothesisTests.jl/stable/.

Mogensen, P. K., A. N. Riseth, J. M. White, T. Holy, and other contributors (2017): *Optim.jl*, https://docs.juliahub.com/Optim/R5uoh/1.2.2/.

Quinn, J. and other contributors (2015): *CSV.jl*, https://csv.juliadata.org/stable/.

Revels, J. (2015): *BenchmarkTools.jl*, https://juliaci.github.io/BenchmarkTools.jl/stable/.

Segal, D. (2020): *MarketData.jl*, https://juliaquant.github.io/MarketData.jl/stable/.

Silverman, B. W. (1986): *Density Estimation for Statistics and Data Analysis*, Chapman & Hall.

Storopoli, J., R. Huijzer, and L. Alonso (2021): *Julia Data Science*, https://juliadatascience.io.

Taylor, J. E. (2006): *QuantileRegressions.jl*, https://github.com/pkofod/QuantileRegressions.jl.

White, J. M., M. Bouchet-Valat, and other contributors (2016): *CategoricalArrays.jl*, https://categoricalarrays.juliadata.org/stable/.

Wooldridge, J. M. (2010): *Econometric Analysis of Cross Section and Panel Data*, MIT Press.

——— (2014): *Introduction to Econometrics*, Cengage Learning.

——— (2019): *Introductory Econometrics: A Modern Approach*, Cengage Learning, 7th ed.

List of Wooldridge (2019) Examples

Index