

Statistics with Julia:

Fundamentals for Data Science, Machine Learning and Artificial Intelligence.

D R A F T

Hayden Klok, Yoni Nazarathy

May 9, 2019

Preface to this DRAFT version

This DRAFT version of our book includes a complete structure of the contents and an almost-complete code-base using Julia 1.0. We hope you find this draft to be a useful resource. Please let us know of any feedback you have. What has helped you? What more you would like to see? What parts do you think can be improved?

*Hayden Klok
Yoni Nazarathy,
May, 2019.*

Preface

The journey of this book began at the end of 2016 when preparing material for a statistics course for The University of Queensland. At the time, the Julia language was already showing itself as a powerful new and applicable tool, even though it was only at version 0.5. For this reason, we chose Julia for use in the course, since, by exposing students to statistics with Julia early on, they would be able to employ Julia for data science, numerical computation and machine learning tasks later in their careers. This choice was not without some resistance from students and colleagues, since back then, as is still now in 2019, in terms of volume, the R-language dominates the world of statistics, in the same way that Python dominates the world of machine-learning. So why Julia?

There were three main reasons: performance, simplicity and flexibility. Julia is quickly becoming a major contending language in the world of data science, statistics, machine learning, artificial intelligence and general scientific computing. It is easy to use like R, Python and Matlab, but due to its type system and just-in-time compilation, it performs computations much more efficiently. This enables it to be fast, not just in terms of run time, but also in terms of development time. In addition, there are many different Julia packages. These include advanced methods for the data-scientist, statistician, or machine learning practitioner. Hence the language has a broad scope of application.

Our goal in writing this book was to create a resource for understanding the fundamental concepts of statistics needed for mastering machine learning, data science and artificial intelligence. This is with a view of introducing the reader to Julia through the use of it as a computational tool. The book also aims to serve as a reference for the data scientist, machine learning practitioner, bio-statistician, finance professional, or engineer, who has either studied statistics before, or wishes to fill gaps in their understanding. In today's world, such students, professionals, or researchers often use advanced methods and techniques. However, one is often required to take a step back and explore or revisit fundamental concepts. Revisiting these concepts with the aid of a programming language such as Julia immediately makes the concepts concrete.

Now, 2.5 years since we embarked on this book writing journey, Julia has matured beyond V1.0, and the book has matured along with it. Julia can be easily deployed by anyone who wishes to use it. However, currently many of Julia's users are hard-core developers that contribute to the language's standard libraries, and to the extensive package eco-system that surrounds it. Therefore, much of the Julia material available at present is aimed at other developers rather than end users. This is where our book comes in, as it has been written with the end-user in mind. The code examples have been deliberately written in a simple format, sometimes at the expense of efficiency and generality, but with the advantage of being easily readable. Each of the code examples aims to convey a specific statistical point, while covering Julia programming concepts in parallel. In a way, the code examples are reminiscent of examples that a lecturer may use in a lecture to illustrate concepts. The content of the book is written in a manner that does not assume any prior statistical knowledge, and in fact only assumes some basic programming experience and a basic understanding of mathematical notation.

The book contains a total of 10 chapters and 3 appendices. The content may be read continuously, or accessed in an ad-hoc manner. The structure is as follows:

Chapter 1 is an introduction to Julia, including its setup, package manager and the main packages

used in the book. The reader is introduced to some basic syntax, and programmatic structure through code examples that aim to illustrate some of the language's features.

Chapter 2 explores basic probability, with a focus on events, outcomes, independence and conditional probability concepts. Several typical probability examples are presented, along with exploratory simulation code.

Chapter 3 explores random variables and probability distributions, with a focus on the use of Julia's `Distributions` package. Discrete, continuous, univariate and multi-variate probability distributions are introduced and explored as an insightful and pedagogical task. This is done through both simulation and explicit analysis, along with the graphing of associated functions of distributions, such as the PMF, PDF, CDF etc.

Chapter 4 momentarily departs from probabilistic notions to focus on data processing, data summary and data visualizations. The concept of the `DataFrame` is introduced as a mechanism for storing heterogeneous data types with the possibility of missing values. Data frames play an integral component of data science and statistics in Julia, just as they do in R and Python. A brief summary of classic descriptive statistics and their application in Julia is also introduced. This is augmented by the inclusion of concepts such as Kernel Density Estimation and the empirical cumulative distribution function. The chapter closes with some basic functionality for working with files.

Chapter 5 introduces general statistical inference ideas. The sampling distributions of the sample mean and sample variance are presented through simulation and analytic examples, illustrating the central limit theorem and related results. Then general concepts of statistical estimation are explored, including basic examples of the method of moments and maximum likelihood estimation, followed by simple confidence bounds. Basic notions of statistical hypothesis testing are introduced, and finally the chapter is closed by touching basic ideas of Bayesian statistics.

Chapter 6 covers a variety of practical confidence intervals for both one and two samples. The chapter starts with standard confidence intervals for means, and then progresses to the more modern bootstrap method and prediction intervals. The chapter also serves as an entry point for investigating the effects of model assumptions on inference.

Chapter 7 focuses on hypothesis testing. The chapter begins with standard t-tests for population means, and then covers hypothesis tests for the comparison of two means. Then, Analysis of Variance (ANOVA) is covered, along with hypothesis tests for checking independence and goodness of fit. The reader is then introduced to power curves. The chapter closes by touching on a seldom looked at property, the distribution of the p -value.

Chapter 8 covers least squares and statistical linear regression models. It begins by covering least squares and then moves onto the linear regression statistical model, including hypothesis tests and confidence bands. Additional concepts of regression are also explored. These include assumption checking, model selection, interactions and more.

Chapter 9 provides an overview of several more advanced machine learning concepts. At onset, the machine learning paradigm of investigating data is introduced. This includes, training, validation and testing. Then the concept of bias and variance in the context of machine learning is introduced. This goes together with presenting ideas of regularization, applied to linear models. The chapter

then moves onto logistic regression and the generalized linear model. Then further supervised learning methods are introduced, including linear classification, random forests, support vector machines and deep neural networks. Then some unsupervised methods are introduced, including k -means and Principal Component Analysis (PCA). The chapter closes with a brief exploration of Markov decision processes and reinforcement learning.

Chapter 10 moves on to stochastic models in applied probability, giving the reader an indication of the strength of stochastic modelling and Monte-Carlo simulation. It focuses on dynamic systems, where Markov chains, discrete event simulation, and reliability analysis are explored, along with several aspects dealing with random number generation.

Appendix A contains a list of many useful items detailing “how to perform … in Julia”, where the reader is directed to specific code examples that detail directly with these items.

Appendix B lists additional language features of the Julia language that were not used by the code examples in this book.

Appendix C lists additional Julia packages dealing with statistics, machine learning, data science and artificial intelligence that were not used in this book.

Whether you are professional, a student, an educator, a researcher or an enthusiast, we hope that you find this book useful. We hope it can expand your knowledge in fundamentals of statistics with a view towards machine learning, artificial intelligence and data science. We further hope that the integration of Julia code and the content that we present help you quickly apply Julia for such purposes.

We would like to thank colleagues, family members and friends for their feedback, comments and suggestions. These include, Milan Bouchet-Valat, Vektor Dewanto, Heidi Dixon, Jaco Du Plessis, Vaughan Evans, Liam Hodgkinson, Bogumił Kamiński, Dirk Kroese, Benoit Liquet, Ruth Luscombe, Geoff McLachlan, Moshe Nazarathy, Robert Salomone, Alex Stenlake, James Tanton and others.

Hayden Klok and Yoni Nazarathy.

Contents

Preface	i
1 Introducing Julia - DRAFT	1
1.1 Language Overview	3
1.2 Setup and Interface	11
1.3 Crash Course by Example	17
1.4 Plots, Images and Graphics	24
1.5 Random Numbers and Monte Carlo	31
1.6 Integration with Other Languages	39
2 Basic Probability - DRAFT	45
2.1 Random Experiments	46
2.2 Working With Sets	57
2.3 Independence	66
2.4 Conditional Probability	67
2.5 Bayes' Rule	69
3 Probability Distributions - DRAFT	75
3.1 Random Variables	75
3.2 Moment Based Descriptors	79
3.3 Functions Describing Distributions	83

3.4	The Distributions Package	90
3.5	Families of Discrete Distributions	96
3.6	Families of Continuous Distributions	106
3.7	Joint Distributions and Covariance	122
4	Processing and Summarizing Data - DRAFT	131
4.1	Data Frames and Cleaning Data	132
4.2	Summarizing Data	137
4.3	Plotting Data	141
4.4	Kernel Density Estimation	145
4.5	Plotting Cumulative Probabilities	147
4.6	Working with Files	151
5	Statistical Inference Ideas - DRAFT	153
5.1	A Random Sample	154
5.2	Sampling from a Normal Population	156
5.3	The Central Limit Theorem	165
5.4	Point Estimation	167
5.5	Confidence Interval as a Concept	179
5.6	Hypothesis Tests Concepts	182
5.7	A Taste of Bayesian Statistics	191
6	Confidence Intervals - DRAFT	201
6.1	Single Sample Confidence Intervals for the Mean	202
6.2	Two Sample Confidence Intervals for the Difference in Means	204
6.3	Bootstrap Confidence Intervals	211
6.4	Confidence Interval for the Variance of Normal Population	214
6.5	Prediction Intervals	218

6.6 Credibility Intervals	220
7 Hypothesis Testing - DRAFT	221
7.1 Single Sample Hypothesis Tests for the Mean	222
7.2 Two Sample Hypothesis Tests for Comparing Means	231
7.3 Analysis of Variance (ANOVA)	236
7.4 Independence and Goodness of Fit	247
7.5 Power Curves	258
8 Linear Regression - DRAFT	267
8.1 Clouds of Points and Least Squares	268
8.2 Linear Regression with One Variable	279
8.3 Multiple Linear Regression	293
8.4 Model Adaptations	298
8.5 Model Selection	306
9 Machine Learning Basics - DRAFT	309
9.1 Training, Validation and Testing	309
9.2 Bias, Variance and Regularization	310
9.3 Logistic Regression and the Generalized Linear Model	313
9.4 Supervised Learning Methods	317
9.5 Unsupervised Learning Methods	327
9.6 Reinforcement Learning and MDP	336
9.7 A Taste of Generational Adversarial Networks	343
10 Simulation of Dynamic Models - DRAFT	345
10.1 Deterministic Dynamical Systems	346
10.2 Markov Chains	350
10.3 Discrete Event Simulation	365

10.4 Models with Additive Noise	371
10.5 Network Reliability	376
10.6 Common Random Numbers and Multiple RNGs	382
Appendix A How-to in Julia - DRAFT	389
A.1 Basics	389
A.2 Text and I/O	392
A.3 Data Structures	393
A.4 Data Frames	397
A.5 Mathematics	398
A.6 Randomness, Statistics and Machine Learning	401
A.7 Graphics	404
Appendix B Additional Julia Features - DRAFT	407
Appendix C Additional Packages - DRAFT	411
Bibliography	419
List of code listings	420
Index	424

Chapter 1

Introducing Julia - DRAFT

Programming goes hand in hand with mathematics, statistics, data science and many other fields. Scientists, engineers, data scientists and statisticians often need to automate computation that would otherwise take too long or be infeasible to carry out. This is for the purpose of prediction, planning, analysis, design, control, visualization or as an aid for theoretical research. Often, general programming languages such as Fortran, C/C++, Java, Swift, C#, Go, JavaScript or Python are used. In other cases, more mathematical/statistical programming languages such as Mathematica, Matlab/Octave, R, or Maple are employed. The process typically involves analyzing the problem at hand, writing code, analyzing behavior and output, re-factoring, iterating and improving the model. At the end of the day, a critical component is speed, specifically, the speed it takes to reach a solution - whatever it may be.

When trying to quantify speed, the answer is not simple. On the one hand, speed can be quantified in terms of how fast a piece of computer code runs, namely *runtime speed*. On the other hand, speed can be quantified in terms of how fast it takes to code, debug and re-factor computer code, namely *development speed*. Within the realm of *scientific computing* and *statistical computing*, compiled low-level languages such as Fortran, C/C++ and the like generally yield fast runtime performance, however require more care in creation of the code. Hence they are generally fast in terms of runtime, yet slow in terms of development time. On the opposite side of the spectrum are mathematically specialized languages such as Mathematica, R, Matlab as well as Python. These typically allow for more flexibility when creating code, hence generally yield quicker development times. However, runtimes are typically significantly slower than what can be achieved with a low-level language. In fact, many of the efficient statistical and scientific computing packages incorporated in these languages are written in low-level languages, such as Fortran or C/C++, which allows for faster runtimes when applied as closed modules.

A practitioner wanting to use a computer for statistical and mathematical analysis often faces a trade-off between run-time and development time. While speed (both development and runtime) is hard to fully and fairly quantify, Figure 1.1 illustrates a schematic view showing general speed trade-offs between languages. As is postulated by this figure, there is a type of a *Pareto optimal frontier* ranging from the C language on one end to the R language on the other. The location of each language on this figure cannot be determined exactly. However, few would disagree that “R is generally faster to code than C” and “C generally runs faster than R”. So what about Julia?

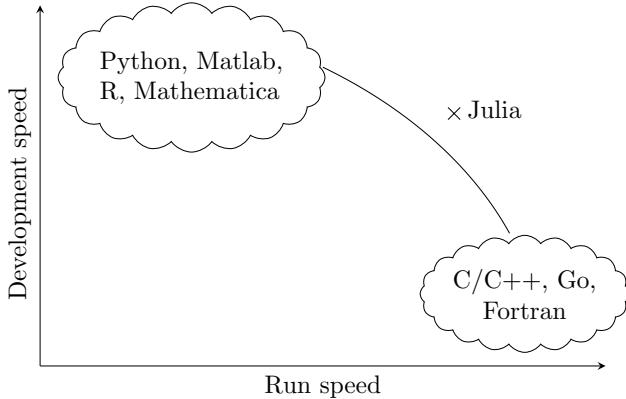


Figure 1.1: A Schematic of run speed vs. development speed.
Observe the Pareto-optimal frontier existing prior to Julia.

The *Julia language and framework* developed in the last several years makes use of a variety of advances in compilation, computer languages, scientific computation and performance optimization. It is a language designed with a view of improving on the previous Pareto-optimal frontier depicted in Figure 1.1. With syntax and style somewhat similar to R, Python and Matlab/Octave, and with performance comparable to that of C/C++ and Fortran, Julia attempts to break the so called *two language problem*. That is, it is postulated that practitioners may quickly create code in Julia, which also runs quickly. Further, re-factoring, improving, iterating and optimizing code can be done in Julia, and does not require the code to be ported to C/C++ or Fortran, since the Julia standard libraries, and almost all of Julia base are written in Julia.

Following this discussion about development speed and run-time speed, we make a rather sharp turn. We focus on *learning speed*. In this context, we focus on learning how to use Julia and in the same process learning and/or strengthening statistical knowledge. In this respect, with the exception of some minor discussions in Section 1.1, “run-time speed and performance” is seldom mentioned in the book. It is rather axiomatically obtained by using Julia. Similarly, coding and complex project development speed is not our focus. Again, the fact that Julia feels like a high-level language, very similar to Python, immediately suggests it is practical to code complex projects quickly in the language. Our focus is on learning quickly.

By following the code examples in this book (there are over 175), we allow you to learn how to use the basics of Julia quickly and efficiently. In the same go, we believe that this book will strengthen your understanding of statistics. In fact, the book contains a self contained overview of elementary probability and statistics, taking the reader through a tour of many concepts, illustrated via Julia code examples. Even if you are a seasoned statistician, data-scientist or probabilist, we are confident that you will find some of our discussions and examples interesting and gain further insight into statistics, machine learning, artificial intelligence and data science as you explore the basics of Julia.

Question: Do I need to have any statistics or probability knowledge to read this book?

Answer: Statistics or probability knowledge is not pre-assumed, however some general mathematics knowledge is assumed. Hence this book is also a self-contained guide for the core principles of probability and statistics. It is ideally suited for a data-scientist wishing to strengthen their core

probability and statistics knowledge while exploring the Julia language.

Question: What experience in programming is needed in-order to use this book?

Answer: While this book is not an introductory programming book, it does not assume that the reader is a professional software developer. Any reader that has coded in some other language (even if only minimally) will be able to follow the code examples in this book and their descriptions.

Question: How to read the book?

Answer: You may either read the book sequentially, or explore ideas and code examples in an ad-hoc manner. In any case, feel free to use the code-repository on GitHub:

<https://github.com/h-Klok/StatsWithJuliaBook>

As you do so, you may want to modify the code in the examples to experiment with various aspects of the statistical phenomena being presented. You may often modify numerical parameters and see what effect your modification has on the output. You may also find the "How-to in Julia" index (Appendix A) useful. This index (also available online) directs you to individual code listings that contain specific examples of "how to".

The remainder of this chapter is structured as follows: In Section 1.1 we present a brief overview of the Julia language. In Section 1.2 we describe some options for setting up a Julia working environment presenting the REPL and JuliaBox. Then in Section 1.3 we dive into Julia code examples designed to highlight basic powerful language features. We continue in Section 1.4 where we present code examples for plotting and graphics. Then in Section 1.5 we overview random number generation and the Monte Carlo method, used throughout the book. We close with Section 1.6 where we illustrate how other languages such as Python, R and C can be easily integrated with your Julia code.

If you are a newcomer to statistics or data-science, then it is possible that many of the examples covered in the first chapter are based on ideas that you have not previously touched. The purpose of the examples is to illustrate key aspects of the Julia language in this context. Hence, if you find the examples of the first chapter overwhelming, feel free to advance to the next chapter where probability is introduced from first principles. The content builds up from there gradually.

1.1 Language Overview

We now embark on a very quick tour of Julia. We start by overviewing language features in broad terms and continue with basic code examples. This section is in no way a comprehensive description of the programming language and its features. Rather, it aims to overview a few select language features, and introduce minimal basics. As a Julia learning resource, this books takes the approach of exploring a variety of examples beginning in Section 1.3, and continuing through Chapters 2 to 9, which include a variety of probability and statistical code examples.

About Julia

Julia is first and foremost a *scientific programming language*. It is perfectly suited for statistics, machine learning, data science and for heavy (as well as light weight) numerical computations. It can also be integrated in user-level applications, however one would not typically use it for front-end interfaces, or game creation. It is an open-source language and platform, visit <https://julialang.org/> for more details. The Julia community brings together contributors from both the scientific computing world and the statistics and data-science world. This puts the Julia language and package system in a good place for combining mainstream statistical methods with methods and trends of the scientific computing world. Coupled with programmatic simplicity similar to Python, and with speed similar to C, Julia is taking an active part of the data-science revolution. In fact, some believe it will become the primary language of data-science in the future (at the moment, most people believe that Python holds this title).

We now discuss a few of the languages main features. If you are relatively new to programming, you may want to skip this discussion, and move to the subsection below which deals with a few basic commands. A key distinction between Julia and other high-level scientific computing languages is that Julia is *strongly typed*. This means that every variable or object has a distinct type that can either explicitly or implicitly be defined by the programmer. This allows the Julia system to work efficiently and integrates well with Julia's *just-in-time (JIT)* compiler. However, in contrast to low level strongly-typed languages, Julia alleviates the user from having to be "type-aware" whenever possible. In fact, many of the code examples in this book, do not explicitly specify types. That is, Julia features *optional typing*, and when coupled with Julia's *multiple dispatch* and *type inference*, Julia's JIT compilation system creates fast running code (compiled to *LLVM*), that is also very easy to program and understand.

The core Julia language imposes very little, and in fact the standard Julia libraries, and almost all of Julia Base, is written in Julia itself. Even primitive operations such as integer arithmetic are written in Julia. The language features a variety of additional packages, some of which are used in this book. All of these packages, including the language and system itself, are free and open sourced (MIT licensed). There are dozens of features of the language that can be mentioned. While it is possible, there is no need to vectorize code for performance. There is efficient support for *Unicode*, including but not limited to UTF-8. C can be called directly from Julia. There are even Lisp-like macros, and other metaprogramming facilities.

Julia development started in 2009 by Jeff Bezanson, Stefan Karpinski, Viral Shah and Alan Edelman. The language was launched in 2012 and has grown significantly since then, with the current version 1.1 as of March 2019. While the language and implementation are open source, the commercial company *Julia Computing* provides services and support for schools, universities, business and enterprises that wish to use Julia. This includes the *Julia Box* service which allows to run Julia via a web browser *Jupyter*. It also supports a free version with enough compute power for all of the examples in this book and more.

A Few Basic Commands

Julia is a full programming language supporting: *procedural programming, object oriented programming, meta-programming, functional programming, numerical computations, network input and*

output, parallel computing and much more. However, when exploring Julia you need to start somewhere. We start with an extended “Hello world”!

Look at the code listing below, and the output that follows. If you’ve programmed previously, you can probably figure out what each of the code lines does. We’ve also added a few comments to this code example, using `#`. Read the code below, and look at the output that follows:

Listing 1.1: Hello world and perfect squares

```

1  println("There is more than one way to say hello:")
2
3  #This is an array consisting of three strings
4  helloArray = ["Hello", "G'day", "Shalom"]
5
6  for i in 1:3
7      println("\t", helloArray[i], " World!")
8  end
9
10 println("\nThese squares are just perfect:")
11
12 #This construct is called a 'comprehension' (or 'list comprehension')
13 squares = [i^2 for i in 0:10]
14
15 #You can loop on elements of arrays without having to use indexing
16 for s in squares
17     print(" ", s)
18 end
19
20 #The last line of every code snippet is also evaluated as output (in addition to
21 #      any figures and printing output generated previously).
22 sqrt.(squares)

```

There is more than one way to say hello:
 Hello World!
 G'day World!
 Shalom World!

These squares are just perfect:
 0 1 4 9 16 25 36 49 64 81 100
 11-element Array{Float64,1}:
 0.0
 1.0
 2.0
 3.0
 4.0
 5.0
 6.0
 7.0
 8.0
 9.0
 10.0

Most of the book contains code listings such as Listing 1.1 above. For brevity, we generally omit comments from code examples, instead, most listings are followed by bullet points highlighting interesting, peculiar and useful code features. Below are some explanatory comments about this listing.

- Line 1 calls the `println()` function for printing the given output string, "There is...hello:".
- Line 4 defines an *array* consisting of 3 strings.
- Lines 6-8 define a *for loop* that executes three times, with the variable `i` incremented on each iteration.
- Line 7, the body of the loop, prints several different arguments. The first, "\t" is a tab spacing. The second is the `i`'th entry of `helloArray` (in Julia array indexing begins with index 1), and the third is an additional string.
- In line 10 the "\n" character is used within the string to signify printing a new line.
- In line 13, a *comprehension* is defined. It consists of the elements, $\{i^2 : i \in \{0, \dots, 10\}\}$. We cover comprehensions further in Listing 1.2.
- Lines 16-18 illustrate that loops may be performed on all elements of an array. In this case, the loop changes the value of the variable `s` to another value of the array `squares` in each iteration. Note the use of the `print()` function to print without a newline.
- Line 22, the last line of the code block applies the `sqrt()` function on each element of the array `squares` by using the '.' broadcast operator. The expression on the last line of every code block, unless terminated by a ";", is presented as output. In this case it is a 11-element array of the number $0, \dots, 10$. Note the type of the elements is `Float64`. We look at types further towards the end of this section.

When exploring statistics and other forms of numerical computation, it is often useful to use a *comprehension* as a basic programming construct. As explained above, a typical form of a comprehension is,

```
[f(x) for x in aaa]
```

Here `aaa` is some array (or more generally, a collection of objects). Such a comprehension creates an array of elements, where each element `x` of `aaa` is transformed via `f(x)`. Comprehensions are ubiquitous in the code examples we present in this book. We often use them due to their expressiveness and simplicity. We now present a simple additional example:

Listing 1.2: Using a comprehension

```
1  array1 = [(2n+1)^2 for n in 1:5]
2  array2 = [sqrt(i) for i in array1]
3  println(typeof(1:5), " ", typeof(array1), " ", typeof(array2))
4  1:5, array1, array2
```

```
UnitRange{Int64}  Array{Int64,1}  Array{Float64,1}
(1:5, [9, 25, 49, 81, 121], [3.0, 5.0, 7.0, 9.0, 11.0])
```

- In line 1 an array is created, named `array1`, which contain the elements of the mathematical set,

$$\{(2n + 1)^2 : n \in \{1, \dots, 5\}\},$$

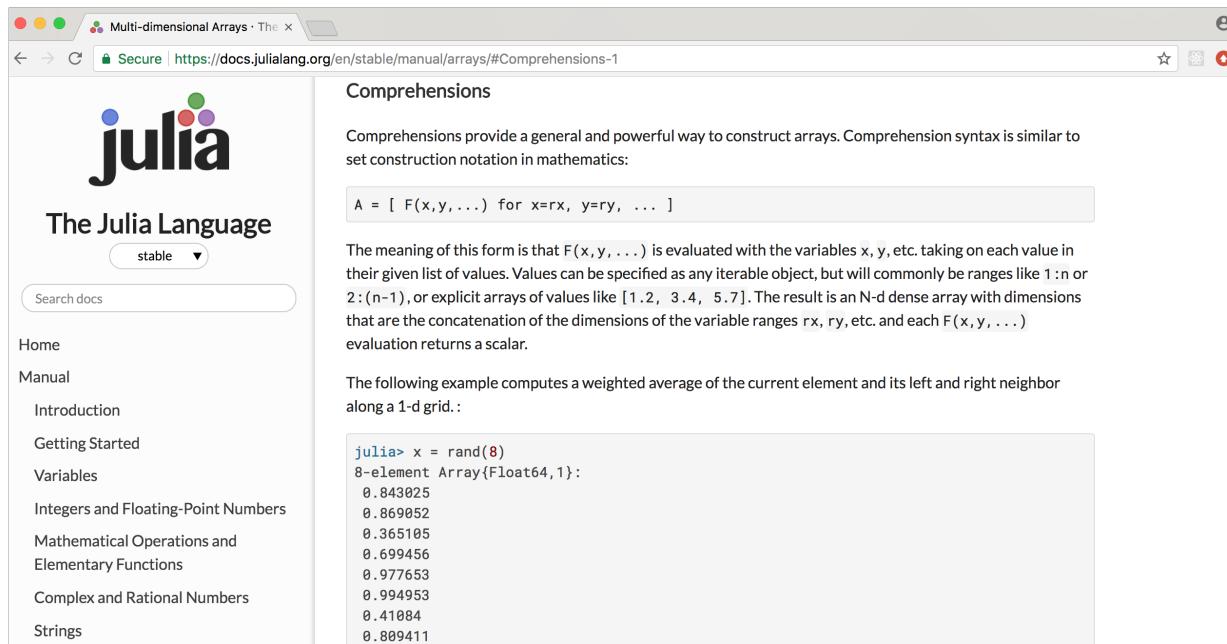


Figure 1.2: Visit <https://docs.julialang.org> for official language documentation.

in order. Note that while mathematical sets are not ordered, arrays generated by Julia comprehensions are ordered. Observe also the literal 2 in the multiplication $2n$, without explicit use of the $*$ symbol.

- In line 2, `array2` is created. An alternative would be to use `sqrt.(array1)`.
- In line 3, we print the `typeof()` three types of expressions. The type of `1:5` (used to create `array1`) is a `UnitRange` of `Int64`. It is a special type of object which encodes “the integers $1, \dots, 5$ ” without explicitly allocating memory for this. Then the types of both `array1` and `array2` are returned. They are both `Array` types, and they contain values of types `Int64` and `Float64` respectively.
- In line 4, a tuple of values is created through the use of a comma between `1:5, array1` and `array2`. As it is the last line of the code it is printed as output. Note in the output that the values of the first array are printed as integers (no decimal point) while the values in the second array are printed as floating point numbers (i.e. contain decimal points).

Getting Help

You may consult the official Julia documentation, <https://docs.julialang.org/> for help. The documentation strikes a balance between precision and readability. See Figure 1.2.

While using Julia, help may be obtained through the use of `?`. For example try, `?sqrt` and you

will see output similar to Figure 1.3.

```
In [1]: ?sqrt
search: sqrt sqrtm isqrt

Out[1]: sqrt(x)

Return  $\sqrt{x}$ . Throws DomainError for negative Real arguments. Use complex negative arguments instead. The prefix operator  $\sqrt{}$  is equivalent to sqrt.
```

Figure 1.3: Snapshot from a Julia Jupyter notebook: Keying in `?sqrt` presents help for the `sqrt()` function.

You may also find it useful to apply the `methods()` function. Try, `methods(sqrt)`. You will see output such as,

```
10 methods for generic function sqrt:

sqrt(x::BigInt) at mpfr.jl:486
sqrt(x::BigFloat) at mpfr.jl:477
sqrt(a::Complex{Float16}) at math.jl:951
sqrt(a::Float16) at math.jl:950
sqrt(x::Float32) at math.jl:426
sqrt(x::Float64) at math.jl:425
sqrt(z::Complex{#s45} where #s45<:AbstractFloat) at complex.jl:392
sqrt(z::Complex) at complex.jl:416
sqrt(x::Real) at math.jl:434
sqrt{T<:Number}(x::AbstractArray{T,N} where N) at deprecated.jl:56
```

This presents different *Julia methods* implementation for the function `sqrt()`. In Julia, a given function may be implemented in different ways depending on different input arguments with each different implementation being a *method*. This is called *multiple dispatch*. Here the various methods of `sqrt()` are shown for different types of input arguments.

Runtime Speed and Performance

While Julia is fast and efficient, for most of this book we don't explicitly focus on runtime speed and performance in this book. Our aim is rather to help the reader learn how to use Julia while enhancing knowledge of probability and statistics. Nevertheless, we now briefly discuss runtime speed and performance.

From a user perspective, Julia feels like an *interpreted language* as opposed to a *compiled language*. With Julia, you are not required to explicitly compile your code before it is run. However, as you use Julia, behind the scenes, the system's JIT compiler compiles every new function and code snippet as it is needed. This often means that on a first execution of a function, runtime is much slower than the second, or subsequent runs. From a user perspective, this is apparent when using other packages (as the example in Listing 1.3 below illustrates, this is often done by the `using` command). On a first call (during a session) to the `using` command of a given package, you may sometimes wait a few seconds for the package to compile. However, afterwards, no such wait is needed.

For day to day statistics and scientific computing needs, you often don't need to give much thought to performance and run speed with Julia. Julia is simply inherently fast! For instance, as we do in dozens of examples in this book, simple Monte Carlo simulations involving 10^6 random variables typically run in less than a second, and are very easy to code. However, as you progress into more complicated projects, many repetitions of the same code block may merit profiling and optimization of the code in question. Hence you may wish to carry out basic profiling.

For basic profiling of performance the `@time` macro is useful. Wrapping code blocks with it (via `begin` and `end`) causes Julia to profile the performance of the block. In Listings 1.3 and 1.4, we carry out such profiling. In both listings, we populate an array, called `data`, containing 10^6 values, where each value is a mean of 500 random numbers. Hence both listings handle half a billion numbers, however, Listing 1.3 is a much slower implementation.

Listing 1.3: Slow code example

```

1  using Statistics
2
3  @time begin
4      data = Float64[]
5      for i in 1:10^6
6          group = Float64[]
7          for j in 1:5*10^2
8              push!(group, rand())
9          end
10         push!(data, mean(group))
11     end
12     println("98% of the means lie in the estimated range: ",
13               (quantile(data,0.01),quantile(data,0.99)) )
14 end;
```

```
98% of the means lie in the estimated range: (0.4699623580817418, 0.5299937027991253)
11.587458 seconds (10.00 M allocations: 8.034 GiB, 4.69% gc time)
```

The actual output of the code gives a range, in this case approximately 0.47 to 0.53 where 98% of the sample means (averages) lie. We cover more on this type of statistical analysis in the chapters that follow.

The second line of output, generated by `@time`, states that it took about 11.6 seconds for the code to execute. There is also further information indicating how many memory allocations took place, in this case about 10 million, totaling just over 8 Gigabytes (in other words Julia writes a little bit, then clears, and repeats this process many times over). This constant read-write is what slows our processing time.

Now look at Listing 1.4 and its output.

Listing 1.4: Fast code example

```

1  using Statistics
2
3  @time begin
4      data = [mean(rand(5*10^2)) for _ in 1:10^6]
5      println("98% of the means lie in the estimated range: ",
6               (quantile(data,0.01),quantile(data,0.99)) )
7 end
```

```
98% of the means lie in the estimated range: (0.469999864362845, 0.5300834606858865)
1.705009 seconds (1.01 M allocations: 3.897 GiB, 10.76% gc time)
```

As can be seen the output gives the same estimate for the interval containing 98% of the means. However, in terms of performance the output of `@time` indicates that this code is clearly superior. It took about 1.7 seconds (compare with 11.6 seconds for Listing 1.3). In this case the code is much faster because far fewer memory allocations are made.

Here are some comments about both code-listings.

- Line 1 (Listing 1.3) calls the `Statistics` package, which is required for the `mean` function.
- Line 4 (Listing 1.3) creates an empty array, `data` of type `Float64`.
- Line 6 (Listing 1.3) creates an empty array, `group`.
- Then lines 7-9 in the same listing loop 500 times, each time pushing to the array, `group`, a new random value generated from `rand()`. The `push!()` function here uses the naming convention of having an exclamation mark when the function modifies the argument. In this case, it modifies `group` by appending another new element. Here is one point where the code is inefficient. The Julia compiler has no direct way of knowing how much memory to allocate for `group` initially, hence some of the calls to `push!()` imply reallocation of the array and copying.
- Line 10 is of a similar nature. The composition of `push!()` and `mean()` imply that the new mean (average of 500 values) is pushed into `data`. However, some of these calls to `push!()` imply a reallocation. At some point, in cases of large i (for example $i \approx 500,000$) the allocated space of `data` will suddenly run out, and at this point the system will need to internally allocate new memory, and copy all 500,000 values to the new location. This is a big cause of inefficiency in our example.
- Line 13 creates a tuple within `println()`, using `(,)`. The two elements of the tuple are return values from the `quantile()` function which compute the 0.01 and 0.99 quantiles of `data`. Quantiles are covered further in Chapters 4 and 5.
- The lines of Listing 1.4 are relatively simpler. All of the computation is carried out in the comprehension on Line 4, within the square brackets `[]`. Writing the code in this way allows the Julia compiler to pre-allocate 10^6 memory spaces for `data`. Then applying `rand()` with an argument of 5×10^2 , indicating the number of desired random values, allows the `rand()` function to operate faster. The functionality of `rand()` is covered in Section 1.5.

Julia is inherently fast, even if you don't give it much thought as a programmer. However, in order to create truly optimized code, one needs to understand the inner workings of the system a bit better. There are some general guidelines that you may follow. A key is to think about memory usage and allocation as in the examples above. Other issues involve allowing Julia to carry out type inference efficiently. Nevertheless, for simplicity, the majority of the code examples of this book ignore types as much as possible. The code is generally fast without considering such issues and in occasional situations, the user may try and optimize it further.

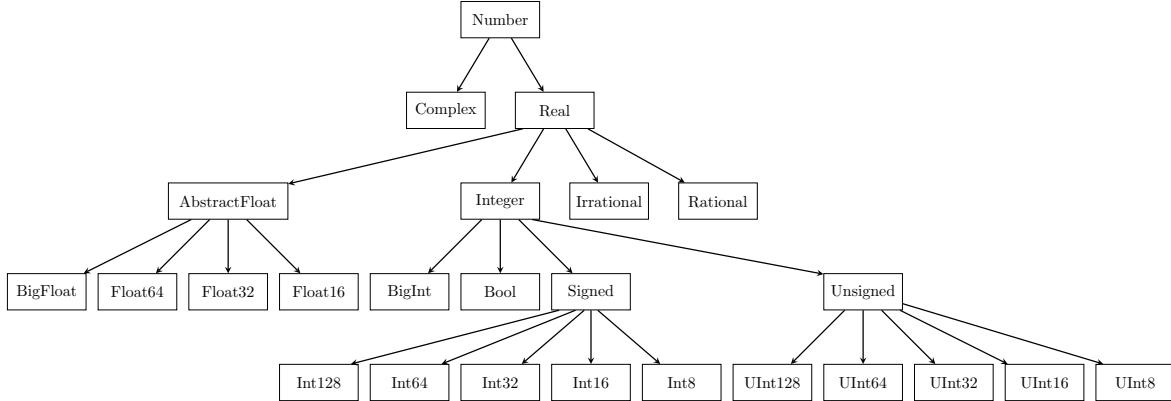


Figure 1.4: Type hierarchy for Julia numbers.

Types and Multiple Dispatch

Functions in Julia are based on the concept of *multiple dispatch*, which means the way a function is executed (i.e. its *method*) is based on the *type* of its inputs (i.e. its *argument types*). Indeed functions can have multiple methods of execution, which can be checked using the `methods()` command.

Julia has a powerful type system which allows for *user defined types*. One can check the type of a variable using the `typeof()` function, while the functions `subtype()` and `supertype()` return the *subtype* and *supertype* of a particular type respectively. As an example `Bool` is a subtype of `Integer`, while `Real` is the supertype of `Integer`. This is illustrated in Figure 1.4, which shows the type hierarchy of numbers in Julia.

One aspect of Julia is that if the user does not specify all variable types in a given piece of code, Julia will attempt to infer what types the unspecified variables should be, and will then attempt to execute the code using these types. This is known as *type inference*, and relies on a type inference algorithm. This makes Julia somewhat forgiving when it comes to those new to coding, and also allows one to quickly mock-up fast working code. It should be noted however that if one wants the fastest possible code, then it is good to specify the types involved. This also helps to prevent *type instability* during code execution.

1.2 Setup and Interface

There are multiple ways to run Julia. Here, we introduce two ways: (1) The *REPL command line interface*, and (2) *JuliaBox* notebooks. We first describe these two alternatives, and then describe the *package manager* which allows to extend Julia's basic functionality by installing additional packages.

No matter where you run Julia, there is an instance of a Julia *kernel* running. The kernel is an instance of the system containing all of the currently compiled functions, loaded packages and defined variables and objects. In complex situations you may even run multiple kernels, sometimes

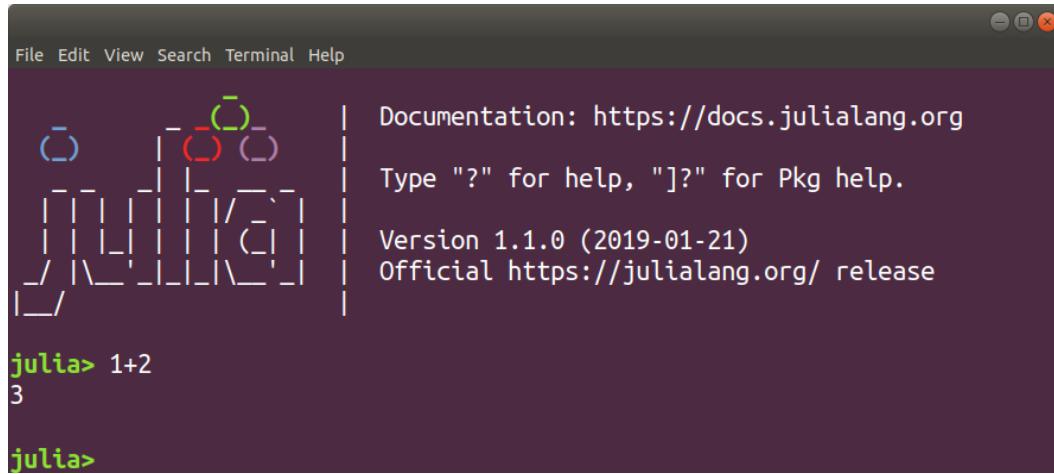


Figure 1.5: Julia’s REPL interface.

in a distributed manner.

REPL Command Line Interface

The *Read Evaluate Print Loop (REPL)* command line interface is a simple and straight forward way of using Julia. It can be downloaded directly from: <https://julialang.org/downloads/>. Downloading it implies downloading the Julia Kernel as well.

Once installed locally, Julia can be launched and the Julia REPL will appear, within which Julia commands can be entered and executed. For example, in Figure 1.5 the code `1+2` was entered, followed by the enter key. Note that if Julia is launched as its own stand alone application, a new Julia instance will appear. However, if you are working in a shell/command-line environment, the REPL can also be launched from within the current environment.

In using the REPL, you will often keep Julia files, such as for example the code listings we have in this book, externally. The standard convention is to name Julia files with a file name ending with `.jl`. In fact, every code listing in this book is available for download from our GitHub page.

JuliaBox

An alternative to using the REPL is to use *JuliaBox*, an on-line product by Julia Computing with a free version available. JuliaBox uses *Jupyter notebooks*. These offer an easy to use web-interface that often serves other languages such as Python and R. See Figure 1.6. Juliabox is available at <https://juliabox.com/>.

The main advantage of JuliaBox is that it can be run from anywhere an internet connection is available. No installation is necessary, and several of the most common packages are preconfigured. It allows users to quickly write, and implement Julia code. It is best suited towards those that need to access Julia quickly, on various machines in multiple locations, and an added benefit is that the

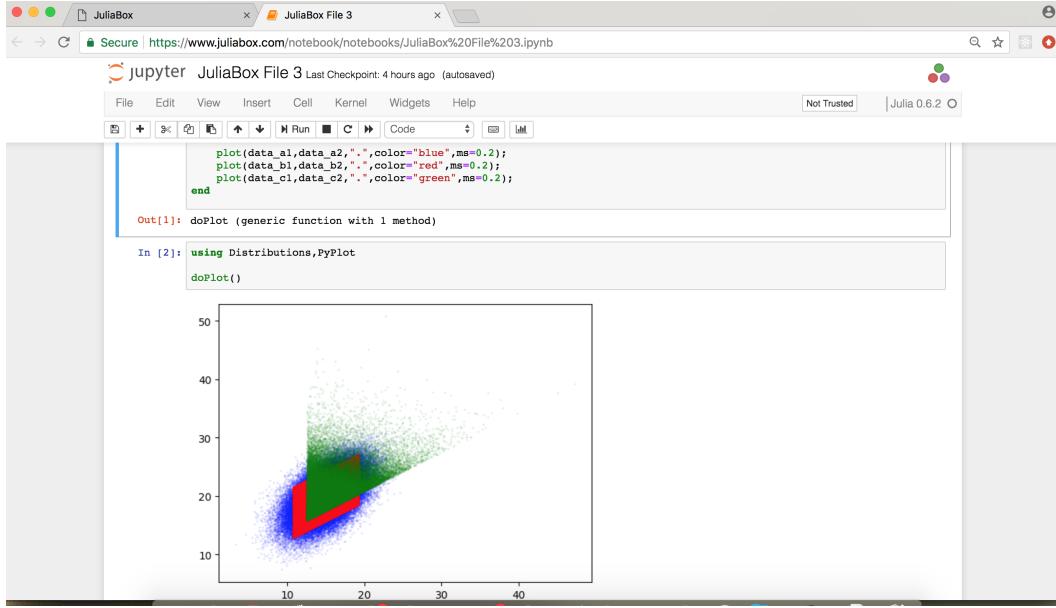


Figure 1.6: An example of a JuliaBox notebook.

computation happens remotely. Juliabox (Jupyter) files can be saved as `*.ipynb` type. In addition notebooks can be exported as PDF as well as other formats.

JuliaBox's notebook interface consists of a series of cells, in which code can be typed and run. Only one cell is active at any time, indicated by the outline around it. When using Julia box, there are two input modes for the keyboard:

Edit Mode: Allows code/text to be entered into the cell. Being in this mode is indicated by a green border around the cell.

Command Mode: Allows notebook level keyboard-activated actions such as toggling line numbering, copying cells etc. It is indicated by a blue border around the selected cell. To enter this mode hit `Esc`.

To run edit mode, simply click on the cell you wish to edit (the active cell will have a green border). To return to command mode press the `esc` key, (the border will turn blue). There are many helpful keyboard shortcuts available. See "Help" at the top of the page, or press `h` while in command mode. Cells can also be different types. The two most useful are:

Code cells: Allows Julia code to be entered and run.

Markdown cells: Allows headings and text paragraphs to be entered using the *Markdown* language including *LAT_EX* formatting.

The nature of a cell can be changed using the dropdown list in the settings at the top of the page, or by pressing `y` or `m` (in command mode), to make it a code cell or markdown cell respectively. Cells can be run by first selecting the cell then pressing `ctrl-enter` or `shift-enter`. Additional

cells can be created by pressing `a` and `b` to create cells above and below respectively. You can see if Juliabox is running if there is an Asterix (*) in the input line [] on the left of the cell, or if there is a fully shaded circle at the top right hand corner of the notebook. You can interrupt a running notebook by selecting `Interrupt Kernel` at the top of the notebook or by pressing `I`.

The Package Manager

Although Julia comes with many built-in commands and features, there is far too much information to be stored in the core program. Instead, Julia relies on packages, which can be added to Julia at your discretion. This allows users to customize their version of Julia depending on their needs, and at the same time offers support for developers who wish to create their own packages-enriching the Julia ecosystem. Note that packages may be either *registered*, meaning that they are part of the Julia package repository; or *unregistered*, meaning they are not.

When using the Julia REPL, you can enter into the Julia *package manager mode* by typing “`]`”. This mode can be exited by typing the backspace key. In this mode, packages can be installed, updated, or removed via the use of specific keywords. The following lists a few of the many useful commands available:

- `]` `add PPPP.jl` adds package PPPP to the current Julia build.
- `]` `status` lists what packages and versions are currently installed.
- `]` `update` updates existing packages.
- `]` `remove PPPP.jl` removes package PPPP from the current Julia build.

When managing packages through JuliaBox, packages can be managed via the Packages icon on the JuliaBox Dashboard. Clicking this icon opens up a dialogue called `Package Builder`, which you can use to add or remove packages via a simple user interface as in Figure 1.7.

As you study the code examples in this book, you will notice that most start with the `using` command, followed by a package name. This is how Julia packages are loaded into the current namespace of the kernel, so that the packages functions, objects and types can be used. Note that writing `using` does not imply installing a package. Installation of a package is a one-time operation which must be performed before the package can be used. In comparison, typing the keyword `using` is required every time a package is loaded into the current namespace.

Packages Used in This Book

The code in this book uses a variety of Julia packages, occasionally introducing useful features. Some of the key packages to use in the context of probability and statistics are, `Distributions`, `DataFrames`, `GLM`, `StatsBase` and `PyPlot` for plotting. However, we also use other packages which provide equally important functionality. A short description of each of the packages that we use in the book is contained below.

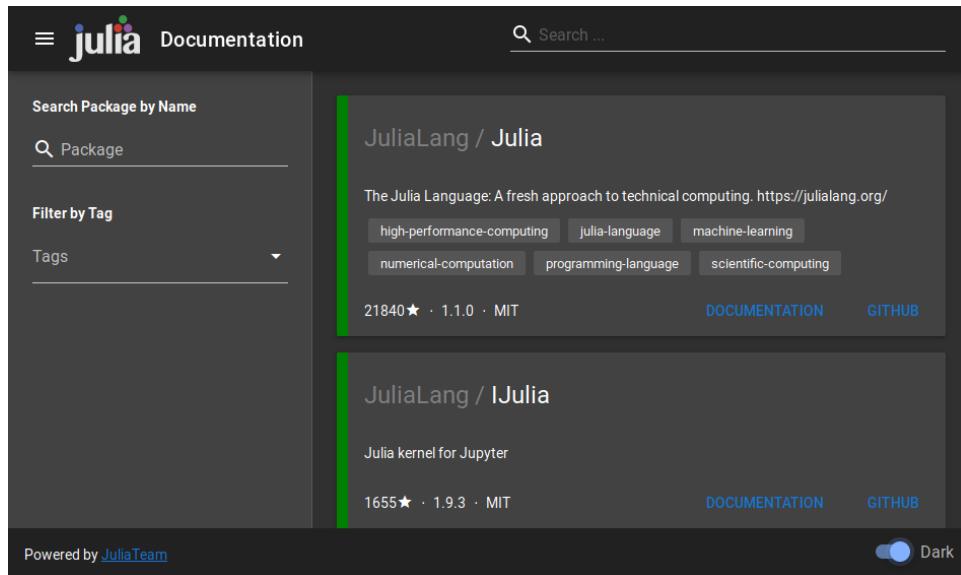


Figure 1.7: The Julia package searcher.

Calculus.jl provides tools for working with basic calculus operations of differentiation and integration both numerically and symbolically.

Clustering.jl provides support for various clustering algorithms.

Combinatorics.jl is a combinatorics library focusing mostly on enumerative combinatorics and permutations.

CSV.jl is a utility library for working with CSV and other delimited files in Julia.

DataFrames.jl is a package for working with tabular data.

DataStructures.jl provides support for various types of data structures.

DecisionTree.jl is a package for decision trees and random forest algorithms.

DifferentialEquations.jl is a suite which provides efficient Julia implementations of numerical solvers for various types of differential equations.

Distributions.jl provides support for working with probability distributions and associated functions.

Flux.jl is a machine learning library written in pure Julia.

GLM.jl is a package on linear models and generalized linear models.

HCubature.jl is an implementation of multidimensional “h-adaptive” (numerical) integration in Julia.

HypothesisTests.jl implements a wide range of hypothesis tests and confidence intervals.

HTTP.jl provides HTTP client and server functionality.

JSON.jl is a package for parsing and printing JSON.

KernelDensity.jl is a kernel density estimation package.

LIBSVM.jl is a package for Support Vector Machines (SVM) using LIBSVM, a general library for SVM.

LightGraphs.jl provides support for the implementation of graphs in Julia.

LinearAlgebra.jl is one of Julia's standard libraries, and provides linear algebra support.

MultivariateStats.jl is a package for multivariate statistics and data analysis, including ridge regression, PCA, dimensionality reduction and more.

NLsolve.jl provides methods to solve non-linear systems of equations in Julia

PyCall.jl provides the ability to directly call and fully interoperate with Python from the Julia language. In this book, we use this mostly for augmenting PyPlot with graphics primitives.

PyPlot.jl provides a Julia interface to the Matplotlib plotting library from Python, and specifically to the matplotlib.pyplot module.

QuadGK.jl provides support for one-dimensional numerical integration using adaptive Gauss-Kronrod quadrature.

Random.jl is one of Julia's standard libraries. It provides support for pseudo random number generation.

RCall.jl provides several different ways of interfacing with R from Julia.

RDatasets.jl provides an easy way to interface with the standard datasets that are available in the core of the R language, as well as several datasets included in many of R's more popular packages.

Roots.jl contains simple routines for finding roots of continuous scalar functions of a single real variable.

SpecialFunctions.jl contains various special mathematical functions, such as Bessel, zeta, digamma, along with sine and cosine integrals, as well as others.

Statistics.jl is one of Julia's standard libraries. It contains functionality for common statistics functions including mean, standard deviation and quantile.

StatsBase.jl provides basic support for statistics by implementing a variety of statistics-related functions, such as scalar statistics, high-order moment computation, counting, ranking, covariances, sampling, and cumulative distribution function estimation.

We are grateful to the dozens of developers that have contributed (and are continuously improving) these great Julia open source packages. You may visit the GitHub page for each of the packages and show your support. Many additional useful packages, not employed in code examples in the book are in Section C.

1.3 Crash Course by Example

Almost every procedural programming language needs functions, conditional statements, loops and arrays. Similarly every scientific programming language needs to support plotting, matrix manipulations and floating point calculations. Julia is no different. To explore these basic programming elements we now present a potpourri of examples. Each example introduces another aspect of the Julia programming language. These examples are not necessarily the minimal examples needed for learning the basics of Julia; they are rather examples exploring what can be done with Julia. If you prefer to first engage with language with shorter and more basic examples, you may follow one of the many Julia tutorials available on the web. One such great resource is the `tutorials/introductory-tutorials/intro-to-julia` folder available with every JuliaBox installation. There you can progress through more than a dozen notebooks showing individual aspects of the language.

Bubble Sort

To begin, let us construct a basic sorting algorithm using first principles. The algorithm we consider here is called *Bubble Sort*. This algorithm takes an input *array*, indexed $1, \dots, n$, and then sorts the elements smallest to largest by allowing the larger elements, or “bubbles”, to “percolate up”. The algorithm is implemented in Listing 1.5. As can be seen from the code, the locations j and $j + 1$ are swapped inside the two *nested loops*. This maintains a non-decreasing order in the array. By using the *conditional statement*, `if`, we check if the numbers at indexes j and $j + 1$ are in increasing order, and if needed, swap them.

Listing 1.5: Bubble sort

```

1  function bubbleSort!(a)
2      n = length(a)
3      for i in 1:n-1
4          for j in 1:n-i
5              if a[j] > a[j+1]
6                  a[j], a[j+1] = a[j+1], a[j]
7              end
8          end
9      end
10     return a
11 end
12
13 data = [65, 51, 32, 12, 23, 84, 68, 1]
14 bubbleSort!(data)
```

8-element Array{Int64,1}:

```

1
12
23
32
51
65
68
84
```

- Line 1 defines a *function*, and its implementation continues up to the `end` statement in line 11. This function receives `a` as an argument; implicitly expected to be an array. It then sorts `a` in place, and returns reference to the array. Note that in this case, the function name ends with ‘!’. This exclamation mark is not part of the Julia language, but rather decorates the name of the function, letting us know that the function argument `a` will be modified (`a` being sorted in place). Indeed in Julia, arrays are *passed by reference*.

As you may infer from the code, arrays are indexed in the range, 1 to the length of the array, obtained by `length()`.

- Line 6, swaps the elements `a[j]` and `a[j+1]`. This is done by the type of assignment of the form `m, n = x, y` which is syntactic short hand for `m=x` followed by `n=y`.
- Line 14 calls the function on `data`. As it is the last line of the code block (and is not followed by a ‘;’) the output is the expression evaluated in that line. In our case it is the sorted array. Note that it has a type `Array{Int64, 1}`, meaning an array of integers. Julia inferred this type automatically. Try changing some of the values in Line 13 to floating points, eg. `[65.0, 51.0 ... (etc)]` and see how the output changes.

Keep in mind that Julia already contains standard sorting functions such `assort()` and `sort!()`, so you don’t need to implement your own sorting function as we did. For more information on these functions use `?sort`. Also, the bubble sort algorithm is not the most efficient sorting algorithm, but is introduced here as a means of understanding Julia better. For an input array of length n , it will run line 5 about $n^2/2$ times. For non-small n , this is much slower performance than optimal sorting algorithms running comparisons only an order of $n \log(n)$ times.

Roots of a Polynomial

Now let us consider a different type of programming example, one that comes from elementary numerical analysis. Consider the polynomial

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0,$$

with real valued coefficients a_0, \dots, a_n . Say we wish to find all x values that solve the equation $f(x) = 0$. We can do this numerically with Julia using the `find_zeros()` function from the `Roots` package. This general purpose solver takes a function as input and numerically tries to find all its roots within some domain. As an example, consider the quadratic polynomial,

$$f(x) = -10x^2 + 3x + 1.$$

Ideally, we would like to supply the `roots` function with the coefficient values, -10 , 3 and 1 . However, `find_zeros()` is not designed for a specific polynomial, but rather for any Julia function that represent a real mathematical function. Hence one way to handle this is to define a Julia function specifically for this quadratic $f(x)$ and give it as an argument to `find_zeros()`. However, here we will take this one step further, and create a slightly more general solution. We first create a function called `polynomialGenerator` which takes a list of arguments representing the coefficients, a_n, a_{n-1}, \dots, a_0 and returns the corresponding polynomial function. We then use this function as an argument to the `roots` function, which then returns the roots of the original polynomial. The code in Listing 1.6 shows our approach.

Once the roots are obtained, we plot the polynomial along with its roots. In our example it is straightforward to solve the roots analytically and verify the code. We do this using the quadratic formula as follows:

$$x = \frac{-3 \pm \sqrt{3^2 - 4(-10)}}{2(-10)} = \frac{3 \pm 7}{20} \quad \Rightarrow \quad x_1 = 0.5, \quad x_2 = -0.2.$$

Listing 1.6: Roots of a polynomial

```

1  using Roots
2
3  function polynomialGenerator(a...)
4      n = length(a)-1
5      poly = function(x)
6          return sum([a[i+1]*x^i for i in 0:n])
7      end
8      return poly
9  end
10
11 polynomial = polynomialGenerator(1,3,-10)
12 zeroVals = find_zeros(polynomial,-10,10)
13 println("Zeros of the function f(x): ", zeroVals)
```

Zeros of the function f(x): [-0.2, 0.5]

Whilst this example is simple, there are quite a few aspects of the Julia programming language that are worth commenting on.

- In line 1 we employ the `using` keyword to indicate to Julia to include elements from the package `Roots`. Note that this assumes the packages have already been added as part of the Julia configuration.
- Lines 3–9 define the function `polynomialGenerator()`. An argument, `a`, along with the splat operator `...` indicates that the function will accept a comma separated list of parameters of unspecified length. For our example we have three coefficients, specified in line 11.
- Line 4 makes use of the `length()` function, reading off how many arguments were given to the function `polynomialGenerator()`. Notice that the degree of the polynomial, represented in the local variable `n` is one less than the number of arguments.
- Lines 5–7 are quite special. They define a new function with an input argument `x`, and that function is stored in the variable `poly` and then returned as an argument. Indeed, one can pass functions as arguments, and store them in variables.
- The main workhorse of this function is line 6, where the `sum()` function is used to sum over an array of values. This array is implicitly defined using a *comprehension*. In this case, the comprehension is `[a[i+1]*x^i for i in 0:n]`. This creates an array of length $n + 1$ where the i 'th element of the array is `a[i+1]*x^i`.
- In line 12 the `find_zeros()` function from the `Roots` package is used to find the roots of the polynomial (the latter arguments are guesses for the roots). The calculated roots are then assigned to `zeroVals`, and in line 13 the output is printed.

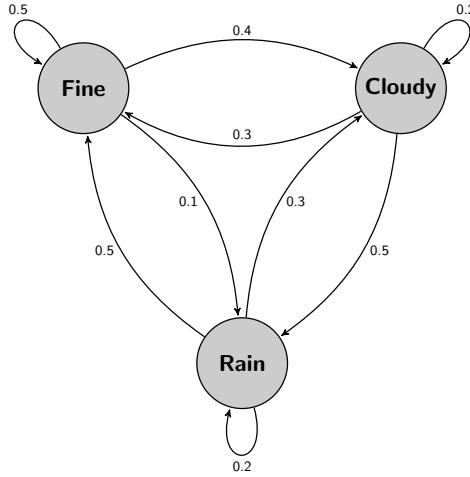


Figure 1.8: Three state Markov chain of the weather.
Notice the sum of the arrows leaving each state is 1.

Steady State of a Markov Chain

We now introduce some basic linear algebra computations and simulation through a simple *Markov chain* example. Consider a theoretical city, where the weather is described by three possible states: (1) ‘Fine’, (2) ‘Cloudy’ and (3) ‘Rain’. On each day, given a certain state, there is a probability distribution for the weather state of the next day. This simplistic weather model constitutes a *discrete time* (homogeneous) Markov chain. This Markov chain can be described by the *Transition Probability Matrix*, P , where the entry $P_{i,j}$ indicates the probability of transitioning to state j given that the current state is i . The transition probabilities are illustrated in Figure 1.8.

One important computable quantity for such a model is the long term proportion of occupancy in each state. That is, in steady state, what proportion of the time is the weather in state 1, 2 or 3. Obtaining this *stationary distribution*, denoted by the vector $\pi = [\pi_1, \pi_2, \pi_3]$ (or an approximation for it) can be achieved in several ways, as shown in Listing 1.7. For pedagogical and exploratory reasons we use the following four methods to solve the stationary distribution:

1. By raising the matrix P to a high power, (repeated matrix multiplication of P with itself), the limiting distribution is obtained in any row. Mathematically,

$$\pi_i = \lim_{n \rightarrow \infty} [P^n]_{j,i} \quad \text{for any index, } j. \quad (1.1)$$

2. We solve the (overdetermined) linear system of equations,

$$\pi P = \pi \quad \text{and} \quad \sum_{i=1}^3 \pi_i = 1. \quad (1.2)$$

This linear system of equations can be reorganized into a system with 3 equations and 3 unknowns by realizing that one of the equations inside $\pi P = \pi$ is redundant.

3. By making use of the known fact (*Perron Frobenius Theorem*) that the eigenvector corresponding to the eigenvalue of maximal magnitude, is proportional to π . We find this eigenvector and normalize it.

4. We run a simple Monte Carlo simulation (see also Section 1.5) by generating random values of the weather according to P , and then take the long term proportions of each state.

Listing 1.7: Steady state of a Markov chain in several ways

```

1  using LinearAlgebra, StatsBase
2
3  # Transition probability matrix
4  P = [0.5 0.4 0.1;
5      0.3 0.2 0.5;
6      0.5 0.3 0.2]
7
8  # First way
9  piProb1 = (P^100)[1,:]
10
11 # Second way
12 A = vcat((P' - I)[1:2,:,:],ones(3)')
13 b = [0 0 1]'
14 piProb2 = A\b
15
16 # Third way
17 eigVecs = eigvecs(copy(P'))
18 highestVec = eigVecs[:,findmax(abs.(eigvals(P)))[2]]
19 piProb3 = Array{Float64}(highestVec)/norm(highestVec,1);
20
21 # Fourth way
22 numInState = zeros(3)
23 state = 1
24 N = 10^6
25 for t in 1:N
26     numInState[state] += 1
27     global state = sample(1:3,weights(P[state,:]))
28 end
29 piProb4 = numInState/N
30
31 [piProb1 piProb2 piProb3 piProb4]
```

The output from listing 1.7 is shown below. Each column represents the stationary distribution obtained from methods 1 to 4.

```

3x4 Array{Float64,2}:
 0.4375  0.4375  0.4375  0.437521
 0.3125  0.3125  0.3125  0.312079
 0.25      0.25      0.25    0.2504
```

The output shows that the four estimates of the vector π that we obtain are very similar. Here are some general comments about the code:

- In lines 4–6 the transition probability matrix P is defined. The notation for defining a matrix in Julia is the same as that of Matlab.
- In line 9, (1.1) is implemented and n is taken as 100, and P^n is calculated in line 10. The first row of the resulting matrix is returned via $[1, :]$. Note that using $[2, :]$ or $[3, :]$ instead will yield (approximately) the same result, since the limit in equation (1.1) is independent of j .

- Lines 12 to 14 use quite a lot of matrix operations to setup the system of equations,

$$\begin{bmatrix} P_{11} - 1 & P_{21} & P_{31} \\ P_{12} & P_{22} - 1 & P_{32} \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \pi_1 \\ \pi_2 \\ \pi_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

These equations are effectively an implementation of equations (1.2). The use of `vcat` (vertical concatenation) creates the matrix on the left hand side by concatenating the 2×3 matrix, $(P' - I)[1:2, :]$ with a row vector of 1's, `ones(3)'` (note the use of `I` which is the identity matrix, with its dimensions determined at compilation time). Finally the solution is given using `A\b` the same as Matlab.

- Lines 17–19 employ the built in `eigvecs` and `eigvals` functions from the `LinearAlgebra` package to find the eigenvalues and a set of eigenvectors of P . Then the `findmax` function is used to find the index matching the eigenvalue with biggest magnitude. Note that `abs` works on complex values as well. Also note that when normalizing in line 21, we use the L_1 norm which is essentially the sum of absolute values of the vector.
- Lines 22–29 carry out a direct Monte Carlo simulation of the Markov chain. Through a million iterations we modify the `state` variable and accumulate the occurrences of each state in line 26. Line 27 is the actual transition, which uses the `sample` function from the `StatsBase` package. At each iteration the next state is randomly chosen based on the probability distribution given the current state. Note that the normalization (from counts to frequency) in Line 29, uses the fact that Julia casts integer counts to floating point numbers upon division. That is, both the variables `numInState` and `N` are an array of integers and an integer respectively, but the division (vector by scalar) makes `piProb4` a floating point array.

Web Interfacing, JSON and String Processing

We now look at a different type of example, dealing with text. Imagine that we wish to analyze the writings of Shakespeare. In particular, we wish to look at the occurrences of some common words in all of his known texts and present a count of a few of the most common words. One simple and crude way to do this is to pre-specify a list of words to count and then specify how many of these words we wish to present.

To add another dimension to this problem, we will use a JSON (*Java Script Object Notation*) file. If you are not familiar with the format of a JSON file, an example is here:

The JSON format uses `{ }` characters to enclose a hierarchical nested structure of key value pairs. In the example above there isn't any nesting, but rather only one top level set of `{,}`. Within that, there are two keys: `words` and `numToShow`. Treating this as a JSON object means that the key `numToShow` has an associated value 5. Similarly, `words` is an array of strings, with each element a potentially interesting word to consider in Shakespeare's texts. In general, JSON files are used for much more complex descriptions of data, but here we use this simple structure for illustration.

Now with some basic understanding of JSON, we can proceed. The code in Listing 1.8 retrieves

Shakespeare's texts from the web and then counts the occurrences of each of the words, ignoring case. We then show a count for each of the numToShow most common words.

Listing 1.8: Web interface, JSON and string parsing

```

1  using HTTP, JSON
2
3  data = HTTP.request("GET",
4    "https://ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.shakespeare.txt")
5  shakespeare = String(data.body)
6  shakespeareWords = split(shakespeare)
7
8  jsonWords = HTTP.request("GET",
9    "https://raw.githubusercontent.com/*"
10   "h-Klok/StatsWithJuliaBook/master/1_chapter/jsonCode.json")
11 parsedJsonDict = JSON.parse(String(jsonWords.body))
12
13 keywords = Array{String}(parsedJsonDict["words"])
14 numberToShow = parsedJsonDict["numToShow"]
15 wordCount = Dict([(x, count(w -> lowercase(w) == lowercase(x), shakespeareWords))
16                   for x in keywords])
17
18 sortedWordCount = sort(collect(wordCount), by=last, rev=true)
19 sortedWordCount[1:numberToShow]

```

```

5-element Array{Pair{String,Int64},1}:
"king"=>1698
"love"=>1279
"man"=>1033
"sir"=>721
"god"=>555

```

- In lines 3-4 `HTTP.request` from the `HTTP` package is used to make a HTTP request.
- In line 5 the body of data is then parsed to a text string via the `String()` function.
- In line 6 this string is then split into an array of individual words via the `split()` function.
- In lines 8-11 the JSON file is first retrieved, and then this string is parsed into a JSON object.
- Line 11 shows the strength of using JSON, the value associated with the JSON key, `words` is accessed. This value (i.e. array of words) is then cast to an `Array{String}` type. Similarly, the value associated with the key `numToShow` is accessed in line 14.
- In line 15 a Julia dictionary is created via `Dict`. It is created from a comprehension of tuples, each with `x` (being a word) in the first element, and the count of these words in `shakespeareWords` as the second element. In using `count` we define the anonymous function as the first argument that compares an input test argument `w` to the given word `x` only in `lowercase`.
- Finally line 18 sorts the dictionary by their values, and line 18 returns (and displays as output) the first most popular `numberToShow` values.

1.4 Plots, Images and Graphics

There are many different plotting packages available in Julia, including PyPlot, Gadfly, as well as several others. There is also the `Plots` package, which acts as a wrapper over several different backends, each with its own strengths and weaknesses. It aims to simplify the process of creating plots, by allowing the user to use a single syntax, regardless of the plotting backend chosen.

In the examples throughout this book, we use the Python plotting package `PyPlot` (`matplotlib`). This was a deliberate decision, as it is mature, flexible, user-friendly, well documented, and its use is already widespread thanks the success of the Python language. Note that in addition to using `PyPlot`, the `PyCall` package is sometimes also called when plotting, as this package allows for Python commands to be called directly, and allows further plotting customization.

There are two main ways to create figures using `PyPlot`, the often simpler procedural API, and the object oriented API, which is more flexible and allows for a high level of customization. Note that the syntax occasionally varies slightly from the `matplotlib` documentation (for example, replacing `"` with `""`).

PyPlot Introduction

We now introduce the package `PyPlot` through several examples in order to show its syntax and how it can be used alongside Julia. In our first example, contained in Listing 1.9, we create a plot of some simple functions, and show various customizable aspects of `PyPlot`.

Listing 1.9: Basic plotting

```

1  using PyPlot
2
3  xGrid = 0:0.1:5
4  G(x) = 1/2*x^2-2*x
5  g(x) = x - 2
6
7  ax = subplot()
8  ax[:spines]["left"][:set_position]("zero")
9  ax[:spines]["right"][:set_position]("zero")
10 ax[:spines]["bottom"][:set_position]("zero")
11 ax[:spines]["top"][:set_position]("zero")
12 ax[:set_xlim](-1,5)
13 ax[:set_ylim](-3,3)
14 ax[:set_aspect]("equal")
15
16 plot(xGrid,G.(xGrid),"b",label="G(x)")
17 plot(xGrid,g.(xGrid),"r",label="g(x)")
18 legend(loc="upper center")
19 title(L"Plot of $G(x)= \frac{1}{2}x^2-2x$ and it's derivative")
20
21 annotate("The minimum", xy=(2, -2), xytext=(3, -2.5), xycoords="data",
22           bbox=Dict("fc"=>"0.8"), arrowprops=Dict("facecolor"=>"black",
23                     "arrowstyle"=>"->"));

```

- Line 1 includes the `PyPlot` package, so that we can create a plot.

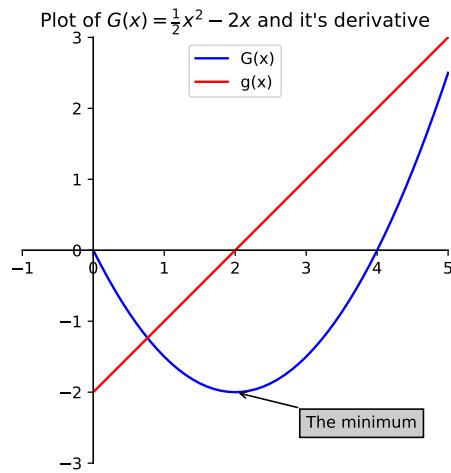


Figure 1.9: An introductory PyPlot example.

- Line 3 sets an xGrid over which our functions will be evaluated.
- Lines 4-5 define our example function $G(x)$, and it's derivative $g(x)$.
- Lines 7-14 are formatted slightly differently as the rest of the code, as they use the object oriented API (this is done for convenience here).
- Line 7 creates a subplot type object.
- Lines 8-11 centre the default borders of the subplot directly on the origin $(0,0)$.
- Lines 12-14 sets the axes limits and enforce a 1:1 scaling.
- Lines 16-17 then plot our functions $G(x)$ and $g(x)$ respectively over xGrid. Simultaneously the line colors are formatted, and label names are specified.
- Line 18 sets the position of the legend.
- Line 19 sets the title and uses `L` as a prefix to the text field, so that it can be formatted using L^AT_EX syntax. Note the use of `$` symbols around mathematical expressions.
- Lines 21-23 create an annotation label, position it, and format its various aspects. Note the use of Dictionary type objects in formatting, and that this format varies slightly from the standard matplotlib documentation.

Histogram of Hailstone Sequence Lengths

We now look at using PyPlot to create a histogram. We construct an example in the context of a well-known mathematical problem. We generate a sequence of numbers as follows: given a positive integer x , if it is even, then the next number in the sequence is $x/2$, otherwise it is $3x + 1$. That is, we start with some x_0 and then iterate $x_{n+1} = f(x_n)$ with

$$f(x) = \begin{cases} x/2 & \text{if } x \bmod 2 = 0, \\ 3x + 1 & \text{if } x \bmod 2 = 1. \end{cases}$$

The sequence of numbers arising from this function is referred to as the *hailstone sequence*. As an example, if $x_0 = 3$, the resulting sequence is

$$3, 10, 5, 16, 8, 4, 2, 1, \dots,$$

where the cycle 4, 2, 1 continues forever. We can then call the length of the sequence the number of steps (possibly infinite) needed to hit 1. Obviously different values of x_0 will result in different hailstone sequences of different lengths.

It is conjectured that, regardless of the x_0 chosen, the sequence will always converge to 1. That is, the length is always finite. However, this has not been proven to date and remains an open question, known as the *Collatz conjecture*. Also, a counter-example hasn't been computationally found. That is, there is no known x_0 for which the sequence doesn't go down to 1.

Now that the context of the problem is set, we create a *histogram* of lengths of hailstone sequences based on different values of x_0 . Our approach is shown in Listing 1.10, where we first create a function which calculates the length of a hailstone sequence based on a chosen value of x_0 . We then use a comprehension to evaluate this function for each value, $x_0 = 2, 3, \dots, 10^7$, and finally plot a histogram of these lengths, shown in Figure 1.10.

Listing 1.10: Histogram of hailstone sequence lengths

```

1  using PyPlot
2
3  function hailLength(x::Int)
4      n = 0
5      while x != 1
6          if x % 2 == 0
7              x = Int(x/2)
8          else
9              x = 3x +1
10         end
11         n += 1
12     end
13     return n
14 end
15
16 lengths = [hailLength(x0) for x0 in 2:10^7]
17
18 plt[:hist](lengths, 1000, normed="true")
19 xlabel("Length")
20 ylabel("Frequency");

```

- Lines 3-14 create the function `hailLength()`, which evaluates the length of a hailstone sequence, n , given the first number in the sequence, x . Notice the usage of `::Int` indicating that this function operates on integer types.
- Line 4 sets n , representing our hailstone sequence length, to zero.

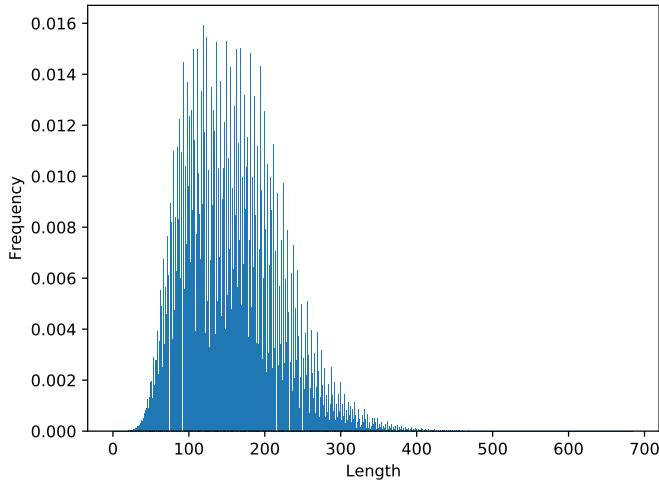


Figure 1.10: Histogram of hailstone sequence lengths.

- Line 5 introduces the concept of a `while` loop, which will sequentially and repeatedly evaluate all code contained within until the specified condition is `false` (i.e. until we obtain a hailstone number of 1). Note the use of the *not-equals comparison operator*, `!=`.
- Line 6 introduces the `mod` operator, `%`, as well as the *equality operator* `==`. They are used in conjunction to check if our current number is even, by returning the remainder of $x/2$ and seeing if this is equivalent to zero (the `==` operator evaluates to a boolean value of either `true` or `false`). If `true`, then we proceed to line 7, else we proceed to line 9.
- Line 11 increase our hailstone sequence length by one each time we generate a new number in our sequence.
- Line 13 returns the length of our generated sequence.
- Line 16 uses a comprehension to evaluate our function for integer values of x_0 between 2 and 10^7 .
- Line 18 plots a histogram of our data, using an arbitrary bin count of 1000.

Graphics Primitives and the `matplotlib Artist API`

We now introduce *graphics primitives* as well as creation of *animation*. For this example, we will use the `animation` and `lines` modules from Python's `matplotlib`. We create a live animation which sequentially draws the edges of a fully-connected mathematical *graph*. This is a mathematical object that consists of *vertices*, which can be thought of as nodes, and *edges*, which can be thought of as lines connecting the vertices.

In this example, given an integer n specifying the number of vertices, we constructs a series equally spaced vertices around a *unit circle*. To add another aspect to this example, we obtain the points around the unit circle by considering the complex numbers,

$$z_n = e^{2\pi i \frac{k}{n}}, \quad \text{for } k = 1, \dots, n.$$

We then use the real and imaginary parts of z_n to obtain the horizontal and vertical coordinates respectively. This distributes n points evenly on the unit circle. The example in Listing 1.11 sequentially draws all possible edges connecting each vertex to all remaining vertices.

Listing 1.11: Animated edges of a graph

```

1  using PyPlot, PyCall
2  @pyimport matplotlib.animation as anim
3  @pyimport matplotlib.lines as line
4
5  function graphCreator(n::Int)
6      vertices = 1:n
7      complexPts = [exp(im*2*pi*k/n) for k in vertices]
8      coords = [(real(p),imag(p)) for p in complexPts]
9      xPts = first.(coords)
10     yPts = last.(coords)
11     edges = []
12     for v in vertices
13         [ push!(edges, (v,u)) for u in (v+1):n ]
14     end
15
16     fig, ax = subplots()
17     xlim(-1.5,1.5)
18     ylim(-1.5,1.5)
19     dots = line.Line2D(xPts, yPts, ls="None", marker="o", ms=20, mec="blue",
20                         mfc="blue")
21     ax[:add_artist](dots)
22
23     function animate(i)
24         u, v = edges[i][1], edges[i][2]
25         xpoints = (xPts[u],xPts[v])
26         ypoints = (yPts[u],yPts[v])
27         ax[:plot](xpoints,ypoints,"r-")
28     end
29
30     ani = [animate(i) for i in 1:length(edges)]
31     anim.ArtistAnimation(fig, ani, interval=5, blit="False", repeat_delay=10)
32 end
33
34 graphCreator(16);

```

- Line 1 specifies the usage of package `PyCall`, so that we can use the `@pyimport` macro.
- Lines 2-3 use the `@pyimport` macro to import the `matplotlib.animation`, and `matplotlib.lines` modules, the latter of which can be used to draw graphics primitives. The modules are given the names `anim` and `line` respectively.
- Lines 5-32 define our function, which constructs the *.gif based on n number of vertices.
- Line 6 defines a `UnitRange` of integers, from 1 to n vertices.
- Line 7 then uses the well-known mathematical expression to place n points around the unit sphere using the formula $e^{\frac{2\pi ik}{n}}$ for $k = 1, \dots, n$. The output is an array of complex numbers, representing points on the *imaginary plane*.

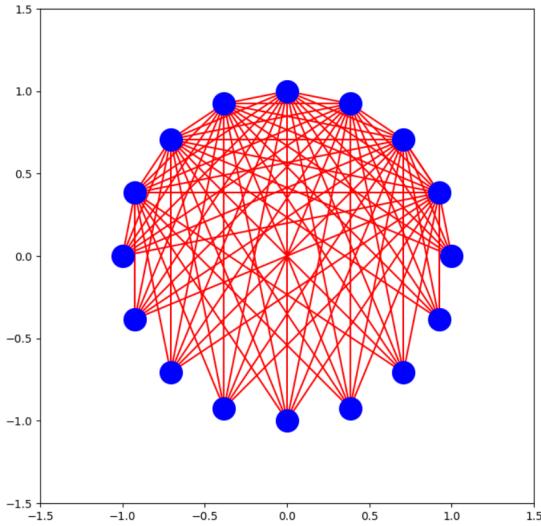


Figure 1.11: Sample frame from the graph animation.

- Line 8 then uses a comprehension to construct an array of tuples (i.e. coordinate pairs) for each vertex.
- Lines 9-10 store these (x, y) coordinates for each vertex in separate arrays.
- Line 11 creates an empty array, which will we will populate with tuples of start-end vertices of each edge.
- Lines 12-14 uses the combination of a for loop and a comprehension to push! tuples of the start-end vertex indices onto the edges array. The setup here prevents “doubling up” on edges (i.e. an edge between vertices 2 and 3 is identical to an edge between 3 and 2).
- In line 16 the PyPlot figure is setup.
- Lines 19-20 then uses matplotlib.lines (defined as line on Line 3) to create a Line2D type object, i.e. a graphics primitive, of a series of markers, each located at the coordinates of a vertex. This primitive is a single graphical object, and is defined as ‘dots’.
- Line 21 then adds ‘dots’ to our figure.
- Lines 23-28 contain an animation function, which is used to draw each subsequent edge in our fully connected graph. Note that i represents the frame number. Importantly, the animation function starts from 0, therefore we use +1 on line 23 to compensate.
- Lines 23-25 constructs a tuple of x coordinates and a tuple of y coordinates based on each vertex pair for each edge (i.e. (x_1, x_2) , (y_1, y_2)). Line 26 then adds a line based on these coordinates to our plot.
- Line 33 runs our function for 16 vertices.

Images

Finally we look at working with images. In the following example, we load a sample image of stars in space and locate the brightest star. Note that the image contains some amount of noise, in particular, the single brightest pixel is located at [168, 192] in column major. Therefore if we wanted to locate the brightest star by a single pixels intensity, we will not identify the correct coordinates. Therefore in order to locate the brightest star we use a simple method of parsing a kernel over the image, which smoothes the results. Once this is done, we then draw a red circle around the brightest points using graphics primitives.

Listing 1.12: Working with images

```

1  using PyPlot, PyCall
2  @pyimport matplotlib.image as image
3  @pyimport matplotlib.patches as patch
4
5  img = image.imread("stars.png")
6  gImg = img[:, :, 1]*0.299 + img[:, :, 2]*0.587 + img[:, :, 3]*0.114
7  rows, cols = size(gImg)
8
9  function boxBlur(image,x,y,d)
10    if x<=d || y<=d || x>=cols-d || y>=rows-d
11      return image[x,y]
12    else
13      total = 0.0
14      for xi = x-d:x+d
15        for yi = y-d:y+d
16          total += image[xi,yi]
17        end
18      end
19      return total/((2d+1)^2)
20    end
21  end
22
23  blurImg = [boxBlur(gImg,x,y,3) for x in 1:cols, y in 1:rows]
24
25  yOriginal, xOriginal = argmax(gImg).I
26  yBoxBlur, xBoxBlur = argmax(blurImg).I
27
28  fig = figure(figsize=(10,5))
29  axO = fig[:add_subplot](1,2,1)
30  axO[:imshow](gImg, cmap="Greys")
31  axO[:add_artist](patch.Circle([xOriginal, yOriginal], 20, fc="none", ec="red", lw=3))
32
33  axS = fig[:add_subplot](1,2,2)
34  axS[:imshow](blurImg, cmap="Greys")
35  axS[:add_artist](patch.Circle([xBoxBlur, yBoxBlur], 20, fc="none", ec="red", lw=3));

```

- Lines 1-3 are very similar to those of Listing 1.11. Here, we load the `PyPlot` and `PyCall` packages, and then import the python `images` and `patches` modules.
- In line 5 we read the image into memory via the `imread` function from pythons `matplotlib` library, and store it as `img`. Since our image is 400×400 pixels, it is stored as a 400×400 array containing `Float32` values. Each cross-section of this array (i.e. 400×400 layer) represents one of the color layers in the following order: Red, Green, Blue, and luminosity.

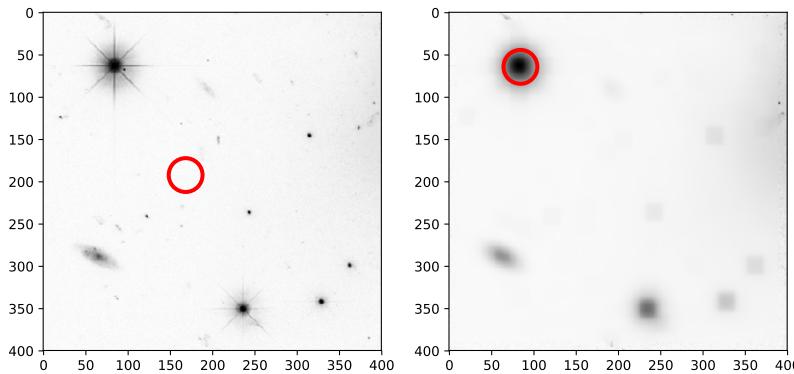


Figure 1.12: Left: Original image.
Right: Smoothed image after noise removal.

- In line 6 we create a greyscale image from the original image data by performing a linear transformation on its RGB layers. The transformation used is a common “Y-Greyscale algorithm”, and the grey image is stored as `gImg`.
- In line 7 the `size()` function is used to determine then number of rows and columns of the array `gImg`. They are then stored as `rows` and `cols` respectively.
- In lines 9-21 the function `boxBlur` is created. This function takes an array of values as input (representing an image), and then passes a kernel over the image data, taking a linear average in the process (this is known as “box blur”). In other words, at each pixel, this function returns a single pixel with a brightness weighting based on the average brightness of the surrounding pixels (or array values) in a given neighborhood within a box of dimensions $2d + 1$. Note that the edges of the image are not smoothed (i.e. a border of un-smoothed pixels of ‘depth’ d exist around the images edges). Visually this kernel smoothing method has the effect of blurring the image.
- In line 23, the function `boxBlur` is parsed over the image for a value of $d = 3$ (i.e. a 5×5 kernel). The smoothed data is then stored as `blurImg`.
- In lines 25 and 26 the `argmax()` function is used to find the index of the pixel with the largest value, for both the non-smoothed image data, and the smoothed image data. Note the use of the trailing “`.I`” at the end of each `argmax`, which extracts the values of the co-ordinates in column-major.
- In lines 28-34 Figure 1.12 is created. The two subplots show the original image vs the smoothed image, and the location of the brightest star for each, located by pixel.

1.5 Random Numbers and Monte Carlo

More than half of the code examples in this book make use of *pseudorandom number generation*, often coupled with the so-called *Monte Carlo simulation method* for obtaining numerical estimates.

We now survey the core ideas and principles of random number generation and Monte Carlo. The main player in this discussion is the `rand()` function. When used without input arguments, `rand()`, generates a “random” number in the interval $[0, 1]$. Questions to now be answered are: How is it random? What does random within the interval $[0, 1]$ really mean? How can it be used as an aid for statistical and scientific computation? For this, let us discuss pseudorandom numbers in a bit more generality.

The “random” numbers we generate using Julia (as well as most “random” numbers used in any other scientific computing platform) are actually pseudorandom - that is, they aren’t really random but rather appear random. For their generation, there is some deterministic (non-random and well defined) sequence, $\{x_n\}$, specified by

$$x_{n+1} = f(x_n, x_{n-1}, \dots), \quad (1.3)$$

originating from some specified *seed*, x_0 . The mathematical function, $f(\cdot)$ is often (but not always) quite a complicated function, designed to yield desirable properties for the sequence $\{x_n\}$ that make it appear random. We wish among other properties for the following to hold:

- (i) Elements x_i and x_j for $i \neq j$ should appear statistically independent. That is knowing the value of x_i should not yield information about the value of x_j .
- (ii) The distribution of $\{x_n\}$ should appear uniform. That is, there shouldn’t be values (or ranges of values) where it appears “more likely” to find values.
- (iii) The range covered by $\{x_n\}$ should be well defined.
- (iv) The sequence should repeat itself as rarely as possible.

Typically, a mathematical function such as $f(\cdot)$ is designed to produce integers in the range $\{0, \dots, 2^\ell - 1\}$ where ℓ is typically 16, 32, 64 or 128 (depending on the number of bits used to represent an integer). In such, we have a sequence of pseudorandom integers. Then if we wish to have a pseudorandom number in the range, $[0, 1]$ (represented via a floating point number), we normalize via,

$$U_n = \frac{x_n}{2^\ell - 1}.$$

When calling `rand()` in Julia (as well as many other programming languages), what we are doing is effectively requesting the system to present us with U_n . Then in the next call, U_{n+1} , and in the call after this U_{n+2} etc... As a user, we don’t care about the actual value of n , we simply trust the computing system that the next pseudorandom number will differ and adhere to the properties (i) - (iv) mentioned above, among others.

Still, the question can be asked, where does the sequence start? For this we have a special name that we call, x_0 , it is the *seed* of the pseudorandom sequence. Typically as a scientific computing system starts up, it sets x_0 to be the current time. This implies that on different startups, x_0, x_1, x_2, \dots will behave differently. However, we may also set the seed ourselves. There are several uses for this and it is often useful for reproducibility of results. The following code listing illustrates setting the seed using Julia’s `Random.seed!()` function.

Listing 1.13: Pseudo random number generation

```

1  using Random
2
3  Random.seed!(1974)
4  println("Seed 1974: ", rand(), "\t", rand(), "\t", rand())
5  Random.seed!(1975)
6  println("Seed 1975: ", rand(), "\t", rand(), "\t", rand())
7  Random.seed!(1974)
8  println("Seed 1974: ", rand(), "\t", rand(), "\t", rand())

```

```

Seed 1974: 0.21334106865797864 0.12757925830167505 0.5047074487066832
Seed 1975: 0.7672833719737708 0.8664265778687816 0.5807364110163316
Seed 1974: 0.21334106865797864 0.12757925830167505 0.5047074487066832

```

As you can see from the output, setting the seed to 1974 produces the same sequence (see lines 1 and 3). However, setting the seed to 1975 produced a completely different sequence.

But why use random or pseudorandom numbers? Sometimes, having arbitrary numbers alleviates programming tasks or helps randomize behavior. For example when designing computer video games, having enemies appear at random spots on the screen yields for a simple implementation. In the context of scientific computing and statistics, the answer lies in the Monte Carlo simulation method. Here the idea is that computations can be aided by repeated sampling and averaging out the result. Many of the code examples in our book do this, below we illustrate one such simple example.

Monte Carlo Simulation

As an example of Monte Carlo, say we wish to estimate the value of π . There are hundreds of known numerical methods to do this and here we explore one. Observe that the area of one quarter section of the unit circle is $\pi/4$. Now if we generate random points, (x, y) , within a unit box, $[0, 1] \times [0, 1]$, and calculate the proportion of total points that fall within the quarter circle, we can approximate π via,

$$\hat{\pi} = 4 \frac{\text{Number of points with } x^2 + y^2 \leq 1}{\text{Total number of points}}.$$

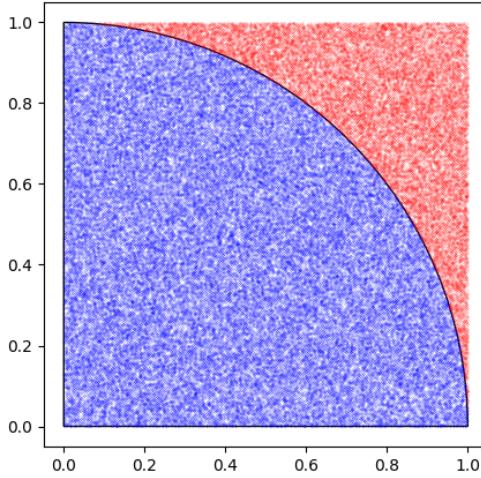
This is performed in Listing 1.14 for 10^5 points. The listing also creates Figure 1.13.

Listing 1.14: Estimating π

```

1  using Random, LinearAlgebra, PyPlot, PyCall
2  @pyimport matplotlib.patches as patch
3  Random.seed!()
4
5  N = 10^5
6  data    = [[rand(), rand()] for _ in 1:N]
7  indata  = filter((x) -> (norm(x) <= 1), data)
8  outdata = filter((x) -> (norm(x) > 1), data)
9  piApprox = 4*length(indata)/N
10 println("Pi Estimate: ", piApprox)
11
12 fig = figure("Primitives", figsize=(5,5))
13 plot(first.(indata), last.(indata), "b.", ms=0.2);

```

Figure 1.13: Estimating π via Monte Carlo.

```

14 plot(first.(outdata),last.(outdata),"r.",ms=0.2);
15 ax = fig[:add_subplot](1,1,1)
16 ax[:set_aspect]("equal")
17 r1 = patch.Wedge([0,0],1,0, 90,fc="none",ec="black")
18 ax[:add_artist](r1)

```

```
Pi Estimate: 3.14068
```

- In Line 3, the seed of the random number generator is set with `Random.seed!()`. This is done to ensure that each time the code is run the estimate obtained is the same.
- In Line 5, the number of repetitions, `N`, is set. Most code examples in this book use `N` as the number of repetitions in a Monte Carlo simulation.
- Line 6 generates an array of arrays. That is, the pair, `[rand(), rand()]` is an array of random coordinates in $[0, 1] \times [0, 1]$.
- Line 7 filters those points to use for the denominator of $\hat{\pi}$. It uses the `filter()` function, where the first argument is an anonymous function, `(x) -> (norm(x) <= 1)`. Here, `norm()` defaults to the L_2 norm, i.e. $\sqrt{x^2 + y^2}$. The resulting `indata` array only contains the points that fall within the unit circle (with each represented as an array of length 2).
- Line 8 does creates the analogous `outdata` array. It doesn't have any value for the estimation, but is rather used for plotting.
- Line 9 calculates the approximation, with `length()` used for the numerator of $\hat{\pi}$ and `N` for the denominator.
- Lines 12-18 are used to create Figure 1.13.

Inside a Simple Pseudorandom Number Generator

The mathematical study of the internals of pseudorandom number generation builds on *number theory*, and related fields and is often not of direct interest for statistics. That is, the specifics of $f(\cdot)$ in (1.3) are rarely the focus. In the sequel, we describe some of the details associated with Julia's `rand()` function, however for exploratory purposes we first attempt to make our own.

A simple to implement class of pseudo-random number generators is the class of *linear congruential generators* (LCG). Here the function $f(\cdot)$ is nothing but an affine (linear) transformation modulo m :

$$x_{n+1} = (a x_n + c) \bmod m. \quad (1.4)$$

The parameters a , c and m are fixed and specify the details of the LCG. Some number theory research has determined “good” values of a and c for given m . For example, for $m = 2^{32}$, setting $a = 69069$ and $c = 1$ yields sensible performance (other possibilities work well, but not all).

In the listing below we generate values based on this LCG, see also Figure 1.14.

Listing 1.15: A linear congruential generator

```

1  using PyPlot
2
3  a, c, m = 69069, 1, 2^32
4  next(z) = (a*z + c) % m
5
6  N = 10^6
7  data = Array{Float64,1}(undef, N)
8
9  x = 808
10 for i in 1:N
11     data[i] = x/m
12     x = next(x)
13 end
14
15 figure(figsize=(12,5))
16 subplot(121)
17 plot(1:1000,data[1:1000],".")
18
19 subplot(122)
20 plt[:hist](data,50, normed = true);

```

- Line 4 defines the function `next()` which implements the right hand side of (1.4).
- Line 7 preallocates an array of `Float64` of length N .
- Line 9 sets the seed at an arbitrary value 808.
- Lines 10-13 loop N times.
- In line 11 we divide the current value of x by m to obtain a number in the range $[0, 1]$. Note that in Julia division of two integers results in a floating point number.
- In line 12 we apply the recursion (1.4) to set a new value for x .

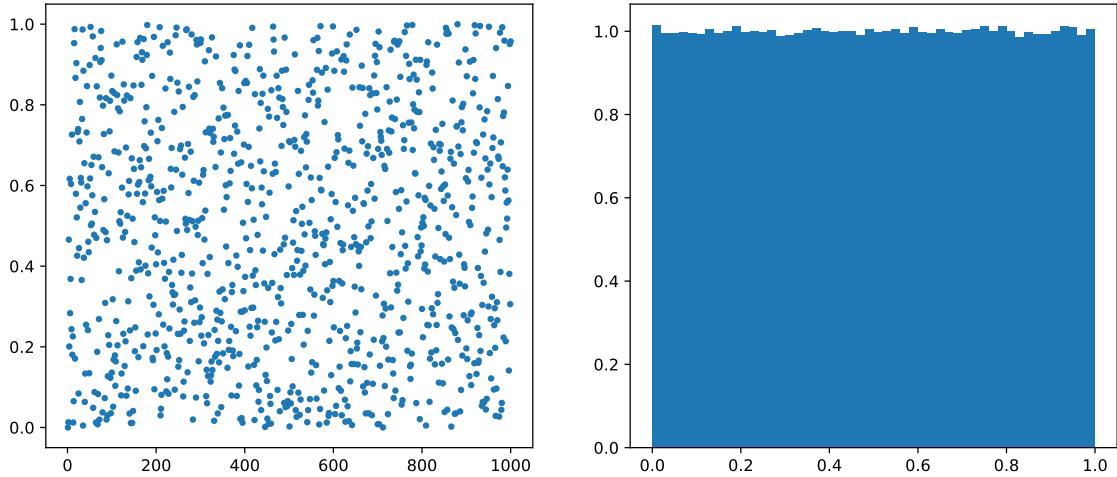


Figure 1.14: Left: The first 1,000 values generated by a linear congruential generator, plotted sequentially. Right: A histogram of 10^6 random values.

- Line 17 plots the first 1000 values of `data`.
- Line 20 creates a histogram of all of `data` with 50 bins. As expected by the theory of LCG, we obtain a uniform distribution.

More About Julia's `rand()`

Having touched the basics, we now describe a few more aspects of Julia's random number generation. The key function at play is `rand()`. However, as you already know, a Julia function may be implemented by different methods; the `rand()` function is no different. To see this, key in `methods(rand)` and you'll see almost 40 different methods associated with `rand()`. Further, if you do this after loading the `Distributions` package into the namespace (by running `using Distributions`) that number will grow beyond 150. Hence in short, you may use `rand()` in many ways in Julia. Throughout the rest of this book, we use it in various ways, including in conjunction with probability distributions, however now we focus on components from the `Base` package.

Some key functions associated with `rand()` are `randn()` for generating normal random variables as well as the following (available after invoking `using Random`): `Random.seed!()`, `randsubseq()`, `randstring()`, `randcycle()`, `bitrand()`, `randperm()`, `shuffle()` and the `MersenneTwister()` constructor. These are discussed in the Julia documentation. You may also use the built-in help to enquire about them. However, let us focus on the constructor `MersenneTwister()` and explain how it can be used in conjunction with `rand()` and variants.

The term *Mersenne Twister* refers to a type of pseudorandom number generator. It is an algorithm that is considerably more complicated than the LCG described above. Generally, its statistical properties are much better than LCG. Due to this, in the past two decades it has made

its way into most scientific programming environments. Julia has adopted it as the standard as well.

Our interest in mentioning the Mersenne Twister is in the fact that we may create an object representing a random number generator implemented via this algorithm. To create such an object, we write for example `rng = MersenneTwister(seed)` where `seed` is some initial seed value. Then the object, `rng`, acts as a random number generator and may serve as (additional) input into `rand()` and related functions. For example, calling `rand(rng)` uses the specific random number generator object passed to it. In addition to `MersenneTwister()`, there are also other ways to create similar objects, such as for example `RandomDevice()`, however we leave it to the reader to investigate these via the online help upon demand.

By creating random number generator objects, you may have more than one random sequence in your application, essentially operating simultaneously. In Chapter 9, we investigate scenarios where this is advantageous from a Monte Carlo simulation perspective. For starters, in the example below we show how a random number generator may be passed into a function as an argument, allowing the function to generate random values using that specific generator.

The example below creates a random path in the plane, starting at $(x, y) = (0, 0)$ and moving up, right, down or left at each step. The movements up ($x += 1$) and right ($y += 1$) are with steps of size 1. However the movements down and left are with steps that are uniformly distributed in the range $[0, 2 + \alpha]$. Hence if $\alpha > 0$, on average the path drifts in the down-left direction. The virtue of this initial example, is that by using *common random numbers* and simulating paths for varying α , we get very different behavior than if we use a different set of random numbers for each path. See Figure 1.15. We discuss more advanced applications of using multiple random number generators in Chapter 9, however we implicitly use this Monte Carlo technique throughout the book, often by setting the seed to a specific value in the code examples.

Listing 1.16: Random walks and seeds

```

1  using PyPlot, Random
2
3  N = 5000
4
5  function path(rng, alpha)
6      x, y = 0.0, 0.0
7      xDat, yDat = [], []
8      for _ in 1:N
9          flip = rand(rng, 1:4)
10         if flip == 1
11             x += 1
12         elseif flip == 2
13             y += 1
14         elseif flip == 3
15             x -= (2+alpha)*rand(rng)
16         elseif flip == 4
17             y -= (2+alpha)*rand(rng)
18         end
19         push!(xDat, x)
20         push!(yDat, y)
21     end
22     return xDat, yDat
23 end
24

```

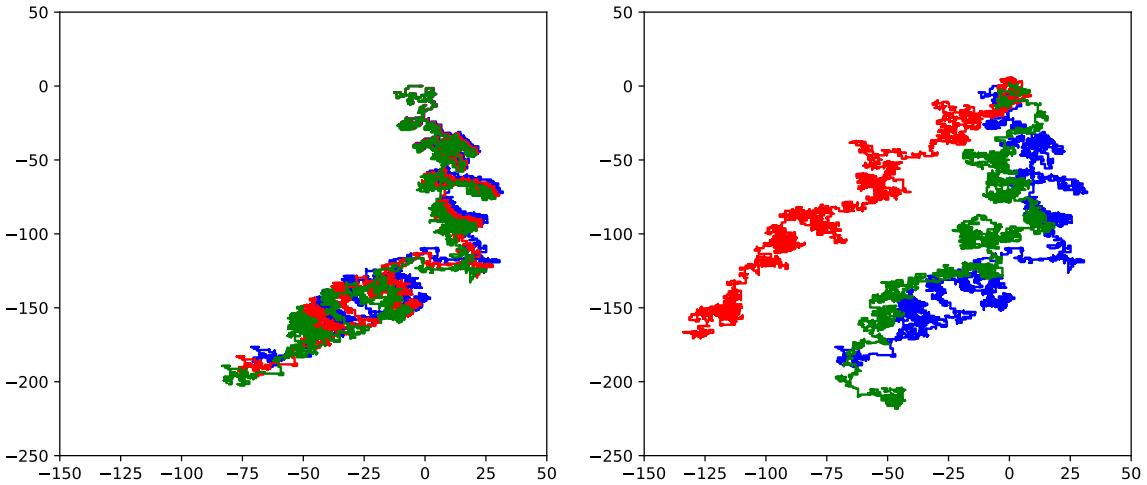


Figure 1.15: Random walks with slightly different parameters.
Left: Trajectories with same seed. Right: Different seed per trajectory.

```

25 alphaRange = 0.2:0.01:0.22
26 figure(figsize=(12,5))
27
28 colors = ["b", "r", "g"]
29 subplot(121)
30 xlim(-150,50)
31 ylim(-250,50)
32 for i in 1:length(alphaRange)
33     xDat, yDat = path(MersenneTwister(27), alphaRange[i])
34     plot(xDat, yDat, color = colors[i])
35 end
36
37 colors = ["b", "r", "g"]
38 subplot(122)
39 xlim(-150,50)
40 ylim(-250,50)
41 rng = MersenneTwister(27)
42 for i in 1:length(alphaRange)
43     xDat, yDat = path(rng, alphaRange[i])
44     plot(xDat, yDat, color = colors[i])
45 end

```

- In Line 3 we set the number of steps each random path is to take, N . Note that it is set as a *global variable* and used by the function that follows. Keep in mind that in more complicated programs it is good practice to avoid using global variables.
- Lines 5-23 define the function `path()`. As a first argument, it takes `rng` (random number generator). That is, the function is designed to receive an object such as `MersenneTwister` as an argument. The second argument is α .
- In lines 8-21 we loop N times each time updating the current coordinate (x and y) and then

pushing the values into the arrays, `xDat` and `yDat`.

- Line 9 generates a random value in the range `1:4`. Observe the use of `rng` as a first argument into `rand()`.
- In lines 15 and 17 we multiply `rand(rng)` by `(2+alpha)`. This creates uniform random variables in the range $[0, 2 + \alpha]$.
- Line 22 returns a tuple, `xDat, yDat`. That is the return value is a tuple of two arrays.
- Lines 32-35 loop 3 times, each time creating a trajectory with a different value of `alpha`. The key point is that as a first argument to `path()` we pass a `MersenneTwister(27)` object. Here 27 is just an arbitrary starting seed. Since we use the same seed on all trajectories, the random values generated within each trajectory are also the same. Hence the effect of varying α can be observed visually.
- Compare with lines 42-45 where the `rng` is defined previously in line 41 and isn't redefined within the loop. This means that each call to `path` in line 43 uses a different set of random values. Notice that the first call, plotted in red uses the same values as in line 33 because the seed in both cases is 27. This explains why the red trajectory is the same for both the left hand and right hand figure.

1.6 Integration with Other Languages

We now illustrate how to interface with the R-language, Python, and C. Note that there are several other packages that enable integration with other languages as well.

Using and calling R Packages

R code, functions, and libraries can be called in Julia via the `RCall.jl` package, which provides several different ways of interfacing with R from Julia. The first way is via the use of “\$”, which can be used to switch between a Julia REPL and an R REPL. Note however that in this case variables are not carried over between the two REPL's. The second way is via the `@rput` and `@rget` macros, which can be used to transfer variables from Julia to the R environment. Finally, the `R"""` (or `@R_str`) macro can also be used to parse R code contained within the string. This macro returns an `RObject` as output (a Julia wrapper type around an R object). Note multi-line strings are also possible via triple string quotations.

We now provide a brief example in Listing 1.17 below. It is related to Listing 7.11 in Chapter 7 carrying out Analysis of Variance (ANOVA). This current example, complements the Chapter 7 examples and makes use of the R `aov` function to calculate the ANOVA f-statistic and p-value of the three machines. Note that this listing assumes that R is already installed.

Listing 1.17: Using R from Julia

```

1  using CSV, DataFrames, RCall
2
3  data1 = CSV.read("machine1.csv", header=false, allowmissing=:none)[:,1]

```

```

4  data2 = CSV.read("machine2.csv", header=false, allowmissing=:none)[:,1]
5  data3 = CSV.read("machine3.csv", header=false, allowmissing=:none)[:,1]
6
7  function R_ANOVA(allData)
8
9      data = vcat([ [x fill(i, length(x))] for (i, x) in
10                  enumerate(allData) ...])
11      df = DataFrame(data, [:Diameter, :MachNo])
12      @rput df
13
14      R"""
15          df$MachNo <- as.factor(df$MachNo)
16          anova <- summary(aov( Diameter ~ MachNo, data=df))
17          fVal <- anova[[1]][["F value"]][[1]][1]
18          pVal <- anova[[1]][["Pr(>F)"]][[1]][1]
19          """
20          println("R ANOVA f-value: ", @rget fVal)
21          println("R ANOVA p-value: ", @rget pVal)
22      end
23
24  R_ANOVA([data1, data2, data3])

```

```

R ANOVA f-value: 10.516968568709089
R ANOVA p-value: 0.00014236168817139574

```

- In line 1 we load the required packages, including RCall.
- In lines 3 to 5 the data from each machine is loaded.
- Lines 7 to 20 contain the main logic of this example. In these lines we create the function R_ANOVA, which takes a Julia array of arrays as input (allData), and outputs the summary results of an ANOVA test carried out in R via the aov function.
- In lines 9 to 10 the array of arrays (allData) are re-arranged into a 2-dimensional array, where the first column contains the observations from each of the arrays, and the second column contains the array index from which each observation has come. The data is re-arranged like this due to the format that the R aov function requires. This re-arrangement is performed via the enumerate function, along with the vcat () function and splat (...) operator.
- In line 11, the data 2-dimensional array data is converted to a DataFrame, and the columns named after the bolt diameter (Diameter) and machine number (MachNo) respectively. The data frame is assigned as df.
- In line 12 the @rput is used to transfer the data frame df to the R workspace.
- In lines 14 to 19 a multi-line R code block is executed inside the R""" macro. First, in line 15, the MachNo column of the R data df is defined as a factor (i.e. is defined as a categorical column) via the R code as.factor() and <- . In line 16 an anova test of the Diameter column of the R data frame df is conducted via the aov function, and parsed to the summary function, with the result stored as anova. In lines 17 and 18, the f-value and p-value is extracted from the anova summary.

- In lines 20 and 21 the f-value and p-value are printed as output. Note the use of the `@rget` which is used to copy the variable from R back to Julia using the same name. The R output shows a calculated f-value of 10.52 and a p-value of 0.00014, which is in agreement with the results obtained from Listing 7.11.

In addition to various R functions, users of R will most likely also be familiar with the R Datasets package, which is a collection of datasets commonly used in statistics. Access to this dataset from Julia is possible via the `RDatasets.jl` package. This package is a collection of 1072 datasets from various packages in R. You can read more about the package in,

<https://vincentarelbundock.github.io/Rdatasets/datasets.html>.

Once installed, datasets can be loaded by specifying first a package name and then a dataset name as arguments to the Julia `datasets()` function. For example, `datasets("datasets", "mtcars")`, will load the `mtcars` dataset from the `datasets` package from `RDatasets`.

Using and Calling Python Packages

It is possible to import Python modules and call python functions directly in Julia via the `PyCall` package. It automatically converts types, and allows data structures to be shared between Python and Julia without copying them.

By default, `add PyCall` uses the `Conda.jl` package to install a minimal Python distribution (via Miniconda) that is private to Julia (not in `PATH`). Further python packages can then be installed from within Julia via `Conda.jl`.

Alternatively, one can use a pre-existing Python installation on the system. In order to do this, one must first set the `python` environment variable to the path of the executable, and then re-build the `PyCall` package. For example, on a system with Anaconda installed, one would use the following from within Julia:

```
] add PyCall
ENV["PYTHON"] = "C:\\Program Files\\Anaconda3\\python.exe"
] build PyCall
```

We now provide a brief example which makes use of the `TextBlob` Python library, which provides a simple API for conducting *Natural Language Processing, (NLP)* tasks, including part-of-speech tagging, noun phrase extraction, sentiment analysis, classification, translation, and more. For our example we will be using `TextBlob` to analyze the sentiment of several sentences. The sentiment analyzer of `TextBlob` outputs a tuple of values, with the first value being the polarity of the sentence (a rating of positive to negative), and the second value a rating of subjectivity (factual to subjective).

In order for Listing 1.18 to work, the `TextBlob` Python library must be installed first. The lines below do just this, however note they must be executed in either a linux shell, or via windows command prompt, not from within a Julia session. (Note that one can swap from the Julia REPL to a shell via “;”).

```
pip3 install -U textblob
python -m textblob.download_corpora
```

Once the Julia Python environment variable is set, and the Python `TextBlob` library has been installed, the Julia code in Listing 1.18 can be executed.

Listing 1.18: NLP via Python `TextBlob`

```
1  using PyCall
2  @pyimport textblob as TB
3
4  str =
5  """Some people think that Star Wars The Last Jedi is an excellent movie,
6  with perfect, flawless storytelling and impeccable acting. Others
7  think that it was an average movie, with a simple storyline and basic
8  acting. However, the reality is almost everyone felt anger and
9  disappointment with its forced acting and bad storytelling."""
10
11 blob = TB.TextBlob(str)
12 [ i[:sentiment] for i in blob[:sentences] ]
```

(0.625, 0.636)
(-0.0375, 0.221)
(-0.46, 0.293)

- In line 1 the `PyCall` function is loaded.
- In line 2 the `pyimport` function is used to call the python library `textblob`, which is then given the alias `TB`.
- In lines 4 to 9, the string `str` is created. For this example, the string is written as a first hand account, and contains many words that give the text a negative tone.
- In line 11 the `TextBlob` function from `TB` (i.e. the alias for `textblob`) is used to parse each sentence in `str`. The resulting ‘text blob’ is stored as `blob`.
- In line 12, a comprehension is used to print the sentiment field for each sentence in `blob`.
- As detailed in the `TextBlob` documentation, the sentiment of the blob is as an ordered pair of polarity and subjectivity, with polarity measured over $[-1.0, 1.0]$ (very negative to very positive), and subjectivity over $[0.0, 1.0]$ (very objective to very subjective). The results indicate that the first sentence is the most positive but is also the most subjective, while the last sentence, is the most negative but also much more objective. The middle sentence is the most neutral, and also the most objective.
- This example only briefly touches on the `PyCall` package, and we encourage the reader to see this packages documentation for further information.

Other Integrations

Julia also allows C and Fortran calls to be made directly via the `ccall` function, which is in Julia base. These calls are made without adding any extra overhead than a standard library call from c code. Note that the code to be called must be available as a shared library. For example, in windows systems, “msvcrt” can be called instead of “libc” (“msvcrt” is a module containing C library functions, and is part of the Microsoft C Runtime Library).

When using the `ccall` function, shared libraries must be referenced in the format `(:function, "library")`. The following is an example where the C function `cos` is called,

```
ccall( (:cos, "msvcrt"), Float64, (Float64,), 0 ).
```

For this example, the `cos` function is called from the `msvcrt` library. Here, `ccall` takes four arguments, the first is the function and library as a tuple, the second is the return type, the third is a tuple of input types (here there is just one), and the last is the input argument (0 in this case).

There are also several other packages that support various other languages as well, such as the `Cxx.jl` or `CxxWrap.jl` packages for C++, `MATLAB.jl` for Matlab, or `JavaCall.jl` for Java. Note that many of these packages are available from <https://github.com/JuliaInterop>.

Chapter 2

Basic Probability - DRAFT

In this chapter we introduce elementary probability concepts. We describe key notions of a probability space along with the concepts of independence and conditional probability. It is important to note that most of the probabilistic analysis carried out in statistics uses random variables and their distributions, and these are introduced in the next chapter. In this chapter however, we focus solely on probability, events and the simple mathematical set-up of a random experiment embodied in a probability space.

The notion of *probability* is the chance of something happening, quantified as a number between 0 and 1 with higher values indicating a higher likelihood of occurrence. However, how do we formally describe probabilities? The standard way to do this is to consider a *probability space*; which mathematically consists of three elements: (1) A *sample space* - the set of all possible outcomes of a certain *experiment*. (2) A collection of *events* - each event is a subset of the sample space. (3) A *probability measure* or *probability function* - which indicates the chance of each possible event occurring.

As a simple example, consider the case of flipping a coin twice. Recall that the sample space is the set of all possible outcomes. We can represent the sample space mathematically as follows,

$$\Omega = \{hh, ht, th, tt\}.$$

Now that the sample space, Ω , is defined, we can consider individual events. For example, let A be the event of getting at least one heads. Hence,

$$A = \{hh, ht, th\}.$$

Or alternately, let B be the event of getting one heads and one tails (where order does not matter),

$$B = \{ht, th\}.$$

There can also be events that consist of a single possible outcome, for example $C = \{th\}$ is the event of getting tails first, followed by heads. Mathematically, the important point is that events are subsets of Ω and often contain more than one outcome. Possible events also include the empty set, \emptyset (nothing happening) and Ω itself (something happening). In the setup of probability, we assume there is a *random experiment* where something is bound to happen.

The final component of a probability space is the probability function. This (mathematically abstract) function, $\mathbb{P}(\cdot)$, takes events as input arguments and returns real numbers in the range $[0, 1]$. It always satisfies $\mathbb{P}(\emptyset) = 0$ and $\mathbb{P}(\Omega) = 1$. It also satisfies the fact that the probability of the union of two disjoint events is the sum of the probabilities and further the probability of the complement of an event is one minus the original probability. More on that in the sequel.

This chapter is structured as follows: In Section 2.1 we explore the basic setup of random experiments with a few examples. In Section 2.2 we explore working with sets in Julia as well as probability examples dealing with unions of events. In Section 2.3 we introduce and explore the concept of independence. In Section 2.4 we move onto conditional probability. Finally, in Section 2.5 we explore Bayes' rule for conditional probability.

2.1 Random Experiments

We now explore a few examples where we set-up a *probability space*. In most examples we present a Monte Carlo simulation of the random experiment and compare results to theoretical ones where available.

Rolling Two Dice

Consider the *random experiment* where two independent, fair, six sided dice are rolled, and we wish to find the probability that the sum of the outcomes of the dice is even. Here the sample space can be represented as $\Omega = \{1, \dots, 6\}^2$, i.e. the *Cartesian product* of the set of single roll outcomes with itself. That is, elements of the sample space are *tuples* of the form (i, j) with $i, j \in \{1, \dots, 6\}$. Say we are interested in the probability of the event,

$$A = \{(i, j) \mid i + j \text{ is even}\}.$$

In this random experiment, since the dice have no inherent bias, it is sensible to assume a *symmetric probability function*. That is, for any $B \subset \Omega$,

$$\mathbb{P}(B) = \frac{|B|}{|\Omega|},$$

where $|\cdot|$ counts the number of elements in the set. Hence for our event, A , we can see from Table 2.1 that,

$$\mathbb{P}(A) = \frac{18}{36} = 0.5.$$

We now obtain this in Julia via both direct calculation and Monte Carlo simulation:

Listing 2.1: Even sum of two dice

```

1 N, faces = 10^6, 1:6
2
3 numSol = sum([iseven(i+j) for i in faces, j in faces]) / length(faces)^2
4 mcEst   = sum([iseven(rand(faces)+rand(faces)) for i in 1:N]) / N
5
6 println("Numerical solution = $numSol \nMonte Carlo estimate = $mcEst")

```

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Table 2.1: All possible outcomes for the sum of two dice. Even sums are shaded.

```
Numerical solution = 0.5
Monte Carlo estimate = 0.499644
```

- In line 1 we set the number of simulation runs, N, and the range of faces on the dice, 1:6.
- In line 3, we use a comprehension to cycle through the sum of all possible combinations of the addition of the outcomes of the two dice. The outcome of the two dice are represented by i, and j respectively, both of which take on the values of faces. We start with i=1, j=1 and add them, and we use the iseven() function to return true if even, and false if not. We then repeat the process for i=1, j=2 and so on, all the way to i=6, j=6. Finally, we count the number of true values by summing all the elements of the comprehension (via sum()), and store the proportion of outcomes which were true in numSol.
- Line 4 uses a comprehension as well. However, in this case we uniformly and randomly select the values which the dice take (akin to rolling them). Again we use the iseven() function to return true if even and false if not, and we repeat this process N times. Using similar logic to line 3, we store the proportion of outcomes which were true in mcEst.
- Line 6 then prints the results using the println() function. Notice the use of \n for creating a newline.

Partially Matching Passwords

We now consider an alphanumeric example. Assume that a password to a secured system is exactly 8 characters in length. Each character is one of 62 possible characters: the letters ‘a’–‘z’, the letters ‘A’–‘Z’ or the digits ‘0’–‘9’.

In this example let Ω be the set of all possible passwords, i.e. $|\Omega| = 62^8$. Now, again assuming a symmetric probability function, the probability of an attacker guessing the correct (arbitrary) password is $62^{-8} \approx 4.6 \times 10^{-15}$. Hence at a first glance, the system seems very secure.

Elaborating on this example, let us also assume that as part of the system’s security infrastructure, when a login is attempted with a password that matches 1 or more of the characters, an event is logged in the system’s security portal (taking up hard drive space). For example, say the original

password is **ax25U1z8**, and a login is attempted using the password **a25XX1z8**. In this case 4 of the characters match and therefore an event is logged.

While the chance of guessing a password and logging in seems astronomically low, in this simple (fictional and overly simplistic) system, there exists a secondary security flaw. That is, hackers may attempt to overload the event logging system via random attacks. If hackers continuously try to log into the system with random passwords, every password that matches one or more characters will log an event, thus taking up more hard-drive space.

We now ask what is the probability of logging an event with a random password? Denote the event of logging a password A . In this case, it turns out to be much more convenient to consider the *complement*, $A^c := \Omega \setminus A$, which is the event of having 0 character matches. We have that $|A^c| = 61^8$ because given any (arbitrary) correct password, there are 61 character options for each character, in order ensure A^c holds. Hence,

$$\mathbb{P}(A^c) = \frac{61^8}{62^8} \approx 0.87802.$$

We then have that the probability of logging an event is $\mathbb{P}(A) = 1 - \mathbb{P}(A^c) \approx 0.12198$. So if, for example, 10^7 login attempts are made, we can expect about 1.2 million login attempts to be written to the security log.

We now simulate such a scenario in Julia:

Listing 2.2: Password matching

```

1  using Random
2  Random.seed!()
3
4  passLength, numMatchesForLog = 8, 1
5  possibleChars = ['a':'z';'A':'Z';'0':'9']
6
7  correctPassword = "3xyZu4vN"
8
9  numMatch(loginPassword) =
10    sum([loginPassword[i] == correctPassword[i] for i in 1:passLength])
11
12 N = 10^7
13
14 passwords = [String(rand(possibleChars, passLength)) for _ in 1:N]
15 numLogs = sum([numMatch(p) >= numMatchesForLog for p in passwords])
16 numLogs, numLogs/N

```

(1219293, 0.1219293)

- In line 2 the seed of the random number generator is set so that the same passwords are generated each time the code is run. This is for reproducibility.
- In line 4 the password length is defined along with the minimum number of character matches before a security log entry is created.
- In line 5 an array is created, which contains all valid characters which can be used in the password. Note the use of ';;', which performs *array concatenation* of the three ranges of characters.

- In line 7 we set an arbitrary correct login password. Note that the type of `correctPassword` is a `String` containing only characters from `possibleChars`.
- In lines 9 to 10 the function `numMatch()` is created, which takes the password of a login attempt, and, via a comprehension, checks each index against that of the actual password. If the index character is correct, it evaluates true, else false. The function then returns how many characters were correct.
- Line 14 uses the functions `rand()` and `String()` along with a comprehension to randomly generate `N` passwords. Note that `String()` is used to convert from an array of single characters to a string.
- Line 15 checks how many times `numMatchesForLog` or more characters were guessed correctly, for each password in our array of randomly generated passwords. It then stores how many times this occurs as the variable `numLogs`.
- Line 16 creates a tuple of both how many login attempts were subsequently logged, and the corresponding proportion of total login attempts. This is also the output of this code listing.

The Birthday Problem

For our next example, we consider the probability of finding a pair of people that share the same birthday in a room. Obviously, ignoring leap years, if there are 366 people present, then it happens with certainty, but what if there are fewer people? Interestingly, with about 50 people, a birthday match is almost certain, and with 23 people in a room, there is about a 50% chance of two people sharing a birthday. At first glance this non-intuitive result is surprising, and hence this famous probability example earned the name *the birthday paradox*. However, we just refer to it as the *birthday problem*.

To carry out the analysis, we assume birthdays are uniformly distributed in the set $\{1, \dots, 365\}$. For n people in a room, we wish to evaluate the probability that at least two people share the same birthday. Set the sample space, Ω , to be composed of ordered tuples (x_1, \dots, x_n) with $x_i \in \{1, \dots, 365\}$. Hence $|\Omega| = 365^n$. Now set the event A to be the set of all tuples (x_1, \dots, x_j) where $x_i = x_j$ for some distinct i and j .

As in the previous example, we consider A^c instead. It consists of tuples where $x_i \neq x_j$ for all distinct i and j (the event of no birthday pair in the group). In this case,

$$|A^c| = 365 \cdot 364 \cdot \dots \cdot (365 - n + 1) = \frac{365!}{(365 - n)!}.$$

Hence we have,

$$\mathbb{P}(A) = 1 - \mathbb{P}(A^c) = 1 - \frac{365 \cdot 364 \cdot \dots \cdot (365 - n + 1)}{365^n}. \quad (2.1)$$

You may compute that for $n = 23$, $\mathbb{P}(A) \approx 0.5073$, and for $n = 50$, $\mathbb{P}(A) \approx 0.9704$.

The code in Listing 2.3 below calculates both the analytic probabilities, as well as estimates them via Monte Carlo simulation. For the numerical solutions, it employs two alternative implementations, `matchExists1()` and `matchExists2()`. The maximum error between the two numerical implementations is presented.

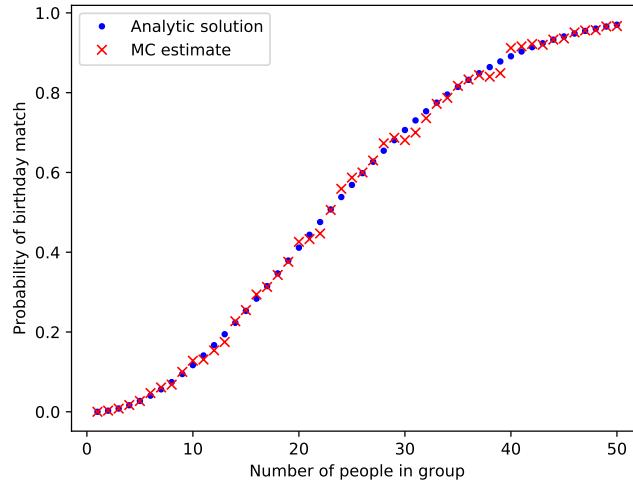


Figure 2.1: Probability that in a room of n people,
at least two people share a birthday.

Listing 2.3: The birthday problem

```

1  using StatsBase, Combinatorics, PyPlot
2
3  matchExists1(n) = 1 - prod([k/365 for k in 365:-1:365-n+1])
4  matchExists2(n) = 1 - factorial(365, 365-big(n))/365^big(n)
5
6  function bdEvent(n)
7      birthdays = rand(1:365, n)
8      dayCounts = counts(birthdays, 1:365)
9      return maximum(dayCounts) > 1
10 end
11
12 probEst(n) = sum([bdEvent(n) for i in 1:N])/N
13
14 xGrid = 1:50
15 analyticSolution1 = [matchExists1(n) for n in xGrid]
16 analyticSolution2 = [matchExists2(n) for n in xGrid]
17 println("Maximum error: $(maximum(abs.(analyticSolution1 - analyticSolution2)))")
18
19 N = 10^3
20 mcEstimates = [probEst(n) for n in xGrid]
21
22 plot(xGrid, analyticSolution1, "b.", label="Analytic solution")
23 plot(xGrid, mcEstimates, "rx", label="MC estimate")
24 xlabel("Number of people in group")
25 ylabel("Probability of birthday match")
26 legend(loc="upper left")

```

Maximum error: 2.4611723650627278208929385e-16

- Lines 3 and 4, each define an alternative functions, `matchExists1()` and `matchExists2()`, for calculating the probability in (2.1). The first implementation uses the `prod()` function

to apply a product over a comprehension. This is in fact a numerically stable way of evaluating the probability. The second implementation evaluates (2.1) in a much more explicit manner. It uses the `factorial()` function from the `Combinatorics` package. Note that the basic `factorial()` function is included in Julia base, however the method with two arguments comes from the `Combinatorics` package. Also, the use of `big()` ensures the input argument is a `BigInt` type. This is needed to avoid overflow for non-small values of n .

- Lines 6-10 define the function `bdEvent()`, which simulates a room full of n people, and if at least two people share a birthday, returns true, otherwise returns false. We now explain how it works.
- Line 7 creates the array `birthdays` of length n , and uniformly and randomly assigns an integer in the range $[1, 365]$ to each index. The values of this array can be thought of as the birth dates of individual people.
- Line 8 uses the function `counts` from the `StatsBase` package to count how many times each birthday occurs in `birthdays`, and assigns these counts to the new array `dayCounts`. The logic can be thought of as follows: if two indices have the same value, then this represents two people having the same birthday.
- Line 9 then checks the array `dayCounts`, and if the maximum value of the array is equal to or greater than one (i.e. if at least two people share the same birthday) then returns true, else false.
- Line 12 defines the function `probEst`, which, when given n number of people, uses a comprehension to simulate N rooms, each containing n people. For each element of the comprehension, (i.e. room), the `bdEvent` function is used to check if at least one birthday pair exists. Then, for each room, the total number of at least one birthday pair is summed up and divided by the total number of rooms N . For large N , the function `probEst` will be a good estimate for the analytic solution of finding at least one birthday pair in a room of n people.
- Lines 14-17, evaluate the analytic solutions over the grid, `xGrid` and prints the maximal absolute error between the solutions. As can be seen from the output, the numerical error is negligible.
- Line 19-20 evaluate the Monte Carlo estimates.
- Lines 22-26 plot the analytic and numerical estimates of these probabilities on the same graph.

Sampling With and Without Replacement

Consider a small pond with a small population of 7 fish, 3 of which are gold and 4 of which are silver. Now say we fish from the pond until we catch 3 fish, either gold or silver. Let G_n denote the event of catching n gold fish. It is clear that unless $n = 0, 1, 2$ or 3 , $\mathbb{P}(G_n) = 0$. However, what is $\mathbb{P}(G_n)$ for $n = 0, 1, 2, 3$? Let us make a distinction between two cases:

Catch and keep - We sample from the population *without replacement*. That is, whenever we catch a fish, we remove it from the population.

Catch and release - We sample from the population *with replacement*. That is, whenever we catch a fish, we return it to the population (pond) before continuing to fish.

The computation of the probabilities $\mathbb{P}(G_n)$ for these two cases of catch and keep, and catch and release, may be obtained via the *Hypergeometric distribution* and *Binomial distribution* respectively. These are both covered in more detail in Section 3.5. We now estimate these probabilities using Monte Carlo simulation. Listing 2.4 below simulates each policy N times, counts how many times zero, one, two and three gold fish are sampled in total, and finally presents these as proportions of the total number of simulations. Note that the total probability in both cases sum to one. The probabilities are plotted in Figure 2.2.

Listing 2.4: Fishing with and without replacement

```

1  using StatsBase, PyPlot
2
3  function proportionFished(gF,sF,numberFished,N,withReplacement = false)
4      function fishing()
5          fishInPond = [ones(Int64,gF); zeros(Int64,sF)]
6          fishCaught = Int64[]
7
8          for fish in 1:numberFished
9              fished = rand(fishInPond)
10             push!(fishCaught,fished)
11             if withReplacement == false
12                 deleteat!(fishInPond, findfirst(x->x==fished, fishInPond))
13             end
14         end
15         sum(fishCaught)
16     end
17
18     simulations = [fishing() for _ in 1:N]
19     proportions = counts(simulations,0:numberFished)/N
20
21     if withReplacement
22         stem(0:numberFished,proportions,basefmt="none",linefmt="r--",
23               markerfmt="rx",label="With replacement");
24     else
25         stem(0:numberFished,proportions,basefmt="none",
26               label="Without replacement")
27     end
28 end
29
30 N = 10^6
31 goldFish, silverFish, numberFished = 3, 4, 3
32
33 figure(figsize=(5,5))
34 proportionFished(goldFish, silverFish, numberFished, N)
35 proportionFished(goldFish, silverFish, numberFished, N, true)
36 ylim(0,0.7)
37 xlabel(L"$n$")
38 ylabel("Probability")
39 legend(loc="upper left");

```

- Lines 3-27 define the function `proportionFished()`, which takes four arguments: the number of gold fish in the pond `gF`, the number of silver fish in the pond `sF`, the number of

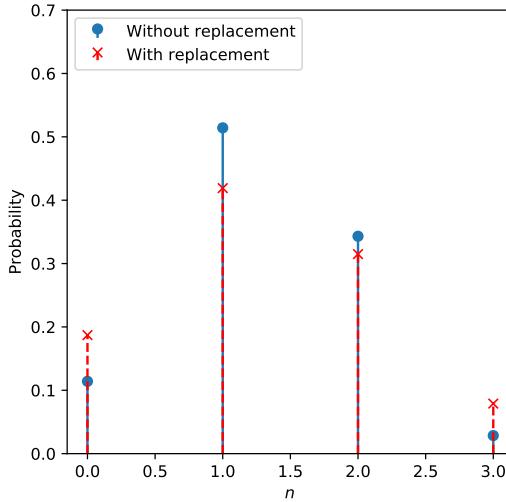


Figure 2.2: Estimated probabilities of catching n of gold fish, with and without replacement.

times we catch a fish numberFished, and a policy of whether we throw back (ie replace) each caught fish, withReplacement, which is set as default to false.

- In lines 4-16 we create an inner function `fishing()` which generates one random instance of a fishing day, returning the number of gold fish caught.
- Line 5 generates an array, where the values in the array represent fish in the pond, with 0's and 1's representing silver and gold fish respectively. Notice the use of the `zeros()` and `ones()` functions, each with a first argument, `Int64` indicating the Julia type.
- Line 6 initializes an empty array, which represents the fish to be caught.
- Lines 8-14 performs the act of fishing `numberFished` times via the use of a `for` loop.
- Lines 9-10 randomly samples a “fish” from our “pond”, and then stores this in value in our `fishCaught` array.
- Line 12 is only run if `false` is used, in which case we “remove” the caught “fish” from the pond. Note that technically we don't remove the exact caught fish, but rather a fish with the same value (0 or 1) via `findfirst()`. This function returns the first index in `fishInPond` with a value equalling `fished`.
- Line 15 is the (implicit) return statement for the function. It sums up how many gold fish were caught (since gold fish are stored as 1's and silver fish as 0's).
- Line 18 implements our chosen policy N times total, with the total number of gold fish each time stored in the array `simulations`.
- Line 19 then uses the `counts` function to return the proportion of times $0, \dots, n$ gold fish were caught.

- Lines 21-27 are used for plotting. Here the `stem()` function from package PyPlot is used to plot the probabilities for specific values of n given a set policy.
- Line 30 sets the total number of simulations.
- Line 31 sets the number of gold and silver fish in the pond, along with the total number of fish we will catch each time.
- Lines 33-39 are used for plotting, and running the function `proportionFished` for our two policies. The output can be seen in Figure 2.2.

Lattice Paths

We now consider a square grid on which an ant walks from the south west corner to the north east corner, taking either a step north or a step east at each grid intersection. This is illustrated in Figure 2.3 where it is clear that there are many possible paths the ant could take. Let us set the sample space to be,

$$\Omega = \text{All possible lattice paths},$$

where the term *lattice path* describes a trajectory of the ant going from the south west point, $(0, 0)$ to the north east point, (n, n) . Since Ω is finite, we can consider the number of elements in it, denoted $|\Omega|$. For a general $n \times n$ grid,

$$|\Omega| = \binom{2n}{n} = \frac{(2n)!}{(n!)^2}.$$

For example if $n = 5$ then $|\Omega| = 252$. The use of the *binomial coefficient* here is because out of the $2n$ steps that the ant needs to take, n steps need to be ‘north’ and n need to be ‘east’.

Within this context of lattice paths there are a variety of questions. One common question has to do with the event (or set):

$$A = \text{Lattice paths that stay above the diagonal the whole way from } (0, 0) \text{ to } (n, n).$$

The question of the size of A , namely $|A|$, has interested many people in combinatorics and it turns out that,

$$|A| = \frac{\binom{2n}{n}}{n+1}.$$

For each counting value of n , the above is called the n ’th *Catalan Number*. For example, if $n = 1$ then $|A| = 1$ and if $n = 3$ then $|A| = 5$. You can try to sketch all possible paths in A for $n = 3$ (there are 5 in total).

So far we have discussed the sample space Ω , and a potential event A . One interesting question to ask deals with the probability of A . That is: *What is the chance that the ant never crosses the diagonal downwards as it journey’s from $(0, 0)$ to (n, n) ?*

The answer to this question depends on the probability measure that we specify (sometimes called a *probability model*). There are infinity many choices for the model and the choice of the right model depends on the context. Here we consider two examples:

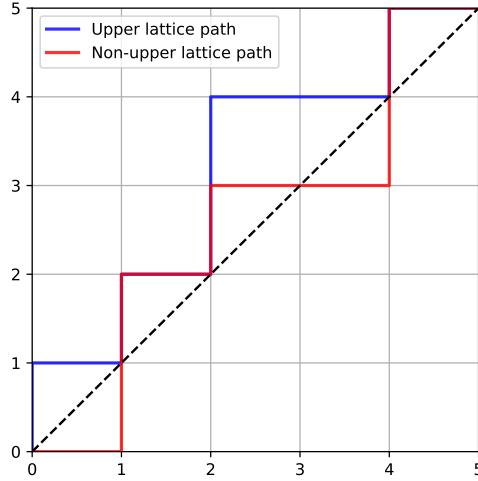


Figure 2.3: Example of two different lattice paths.

Model I - As in the previous examples, assume a symmetric probability space, i.e. each lattice path is equally likely. For this model, obtaining probabilities is a question of counting and the result just follows the combinatorial expressions above:

$$\mathbb{P}_I(A) = \frac{|A|}{|\Omega|} = \frac{1}{n+1}. \quad (2.2)$$

Model II - We assume that at each grid intersection where the ant has an option of where to go ('east' or 'north'), it chooses either east or north, both with equal probability $1/2$. In the case where there is no option for the ant (i.e. it hits the east or north border) then it simply continues along the border to the final destination (n,n) . For this model, it isn't as simple to obtain an expression for $\mathbb{P}(A)$. One way to do it is by considering a *recurrence relation* for the probabilities (sometimes known as *first step analysis*). We omit the details and present the result:

$$\mathbb{P}_{II}(A) = \frac{|A|}{|\Omega|} = \frac{\binom{2n-1}{n}}{2^{2n-1}}.$$

Hence we see that the probability of the event, depends on the probability model used - and this choice is not always a straightforward and obvious one. For example, for $n = 5$ we have,

$$\mathbb{P}_I(A) = \frac{1}{6} \approx 0.166, \quad \mathbb{P}_{II}(A) = \frac{126}{512} \approx 0.246.$$

We now verify these values for $\mathbb{P}_I(A)$ and $\mathbb{P}_{II}(A)$ by simulating both Model I and Model II in the Listing 2.5 below. The listing also creates Figure 2.3.

Listing 2.5: Lattice paths

```

1  using Random, Combinatorics, PyPlot
2  Random.seed!(1)
3
4  n, N = 5, 10^5
5  function isUpperLattice(v)
6      for i in 1:Int(length(v)/2)
7          sum(v[1:2*i-1]) >= i ? continue : return false && break
8      end
9      return true
10 end
11 function plotPath(v,l,c)
12     x,y = 0,0
13     graphX, graphY = [x], [y]
14     for i in v
15         if i == 0
16             x += 1
17         else
18             y += 1
19         end
20         push!(graphX,x), push!(graphY,y)
21     end
22     plot(graphX,graphY,alpha=0.8,label=l, lw=2, c=c)
23 end
24 omega = unique(permuations([zeros(Int,n);ones(Int,n)]))
25 A = omega[isUpperLattice.(omega)]
26 Ac = setdiff(omega,A)
27 figure(figsize=(5,5))
28 plotPath(rand(A), "Upper lattice path", "b")
29 plotPath(rand(Ac), "Non-upper lattice path", "r")
30 legend(loc="upper left");
31 plot([0, n], [0,n], ls="--", "k")
32 xlim(0,n)
33 ylim(0,n)
34 grid("on")
35 pA_modelI = length(A)/length(omega)
36 function randomWalkPath(n)
37     x, y = 0, 0
38     path = []
39     while x<n && y<n
40         if rand()<0.5
41             x += 1
42             push!(path,0)
43         else
44             y += 1
45             push!(path,1)
46         end
47     end
48     if x < n
49         append!(path, zeros(Int64, n-x))
50     else
51         append!(path, ones(Int64, n-y))
52     end
53     return path
54 end
55 pA_modelIItest = sum([isUpperLattice(randomWalkPath(n)) for _ in 1:N])/N
56 pA_modelI, pA_modelIItest

```

```
(0.1666666666666666, 0.24695)
```

- In the code a path is encoded by a sequence of 0 and 1 value, indicating “move east” or “move north” respectively.
- The function `isUpperLattice()` defined in lines 5–10 checks if a path is a lattice path by summing all the odd partial sums, and returning false if any sum ends up at a coordinate below the diagonal. Note the use of the `? :` operator. Also note that on line 6, `Int()` is used to convert the division `length(v)/2` to an integer type.
- The function `plotPath()` defined on lines 11–23 plots a path with a specified label. The code on lines 12–21 generates the horizontal and vertical coordinates of the path, each time adding another coordinate on line 20.
- On line 24, a collection of all possible lattice paths is created by applying the `permutations()` function from the `Combinatorics` package to an initial array of n zeros and n ones. The `unique` function is then used to remove all duplicates.
- In line 25 the `isUpperLattice` function is then applied to each element of `omega` via the `.` operator just after the function name. The result is a boolean array. Then `omega[]` selects the indices of `omega` where the value is true.
- Notice the use of `ls=" - "` in the plotting of the diagonal on line 31.
- In line 35 `pA_modelI` is computed as in (2.2).
- In lines 36–54 the function `randomWalkPath()` is implemented, which creates a random path according to Model II. Note that the code in lines 48–52 appends either zeros or ones to the path, depending on if it hit the north boundary or east boundary first.
- With such a path generation function at hand, estimating a Monte Carlo estimate of the probability is straightforward (line 55).
- Finally, the output line 56 yields values in agreement with the calculations above.

2.2 Working With Sets

As evident from the examples in Section 2.1 above, mathematical *sets* play an integral part in the evaluation of probability models. Any subset of the sample space Ω is also called an *event*. By carrying out *intersections*, *unions* and *differences* of sets, we may often express more complicated events based on smaller ones.

A set is an unordered collection of unique *elements*. A set A is a *subset* of the set B if every element that is in A is also an element of B . The *union* of two sets, A and B , denoted $A \cup B$ is the set of all elements that are either in A or B or both. The *intersection* of the two sets, denoted $A \cap B$, is the set of all elements that are in both A and B . The *difference*, denoted $A \setminus B$ is the set of all elements that are in A but not in B .

In the context of probability, the sample space Ω is often considered as the *universal set*. This allows us to then consider the *complement* of a set A , denoted A^c , which can be constructed via all elements of Ω that are not in A . Note that $A^c = \Omega \setminus A$. Also observe that in the presence of a universal set: $A \setminus B = A \cap B^c$.

Representing Sets in Julia

Julia includes built in capability for working with sets. Unlike an Array, a Set is an unordered collection of unique objects. The simple Listing 2.6 below illustrates how to construct a Set in Juila, and illustrates the use of the union, intersect, setdiff, issubset and in functions.

Listing 2.6: Basic set operations

```

1  A = Set([2,7,2,3])
2  B = Set(1:6)
3  omega = Set(1:10)
4
5  AunionB = union(A, B)
6  AintersectionB = intersect(A, B)
7  AdifferenceB = setdiff(B,A)
8  Bcomplement = setdiff(omega,B)
9  AsymDifferenceB = union(setdiff(A,B),setdiff(B,A))
10 println("A = $A, B = $B")
11 println("A union B = $AunionB")
12 println("A intersection B = $AintersectionB")
13 println("B diff A = $AdifferenceB")
14 println("B complement = $Bcomplement")
15 println("A symDifference B = $AsymDifferenceB")
16 println("The element '6' is an element of A: $(in(6,A))")
17 println("The symmetric difference and the intersection are subsets union: ",
18      issubset(AsymDifferenceB,AunionB),", ", issubset(AintersectionB,AunionB))

```

```

A = Set([7, 2, 3]), B = Set([4, 2, 3, 5, 6, 1])
A union B = Set([7, 4, 2, 3, 5, 6, 1])
A intersection B = Set([2, 3])
B diff A = Set([4, 5, 6, 1])
B complement = Set([7, 9, 10, 8])
A symDifference B = Set([7, 4, 5, 6, 1])
The element '6' is not an element of A: false
The symmetric difference and the intersection are subsets union: true, true

```

- Lines 1-3 create three different sets via the Set function. Note that A contains only three elements, since sets are meant to be a collection of unique elements. Also note that unlike arrays order is not preserved.
- Lines 5-9 perform various operations using the sets created.
- Lines 10-18 create the output seen just below Listing 2.6.

The Probability of a Union

Consider now two events (sets) A and B . If $A \cap B = \emptyset$, then $\mathbb{P}(A \cup B) = \mathbb{P}(A) + \mathbb{P}(B)$. However more generally, when A and B are not *disjoint*, the probability of the *intersection*, $A \cap B$ plays a role. For such cases the *inclusion exclusion formula* is useful:

$$\mathbb{P}(A \cup B) = \mathbb{P}(A) + \mathbb{P}(B) - \mathbb{P}(A \cap B). \quad (2.3)$$

To help illustrate this, consider the simple example of choosing a random lower case letter, ‘a’-‘z’. Let A be the event that the letter is a vowel (one of ‘a’, ‘e’, ‘i’, ‘o’, ‘u’). Let B be the event that the letter is one of the first three letters (one of ‘a’, ‘b’, ‘c’). Now since $A \cap B = \{\text{'a}'\}$, a set with one element, we have,

$$\mathbb{P}(A \cup B) = \frac{5}{26} + \frac{3}{26} - \frac{1}{26} = \frac{7}{26}.$$

For another similar example, consider the case where A is the set of vowels as before, but $B = \{\text{'x}', \text{'y}', \text{'z'}\}$. In this case, since the intersection of A and B is empty, we immediately know that $\mathbb{P}(A \cup B) = 8/26 \approx 0.3077$. While this example is elementary, we now use it to illustrate a type of conceptual error that one may make when using Monte Carlo simulation.

Consider the code listing below, and compare `mcEst1` and `mcEst2` from lines 10 and 11 respectively. Both variables are designed to be estimators of $\mathbb{P}(A \cup B)$. However, one of them is a correct estimator and the other one faulty. In the following we look at the output given from both, and explore the fault in the underlying logic.

Listing 2.7: An innocent mistake with Monte Carlo

```

1  using Random, StatsBase
2  Random.seed!(1)
3
4  A = Set(['a', 'e', 'i', 'o', 'u'])
5  B = Set(['x', 'y', 'z'])
6  omega = 'a':'z'
7
8  N = 10^6
9
10 println("mcEst1 \t \t mcEst2")
11 for _ in 1:N
12     mcEst1 = sum([in(sample(omega), A) || in(sample(omega), B) for _ in 1:N]) / N
13     mcEst2 = sum([in(sample(omega), union(A, B)) for _ in 1:N]) / N
14     println(mcEst1, "\t", mcEst2)
15 end

```

First observe line 12. In Julia, `||` means “or”, so at first glance the estimator `mcEst1` looks sensible, since:

$$A \cup B = \text{the set of all elements that are in } A \text{ or } B.$$

Hence we are generating a random element via `sample(omega)` and checking if it is an element of A or an element of B . However there is a subtle error. Each of the N random experiments involves two separate calls to `sample(omega)`. Hence the code on line 12 simulates a situation where conceptually, the sample space, Ω is composed of 2-tuples, not single letters!

Hence the code computes probabilities of the event, $A_1 \cup B_2$ where,

$$\begin{aligned} A_1 &= \text{First element of the tuple is a vowel,} \\ B_2 &= \text{Second element of the tuple is an xyz letter.} \end{aligned}$$

Now observe that A_1 and B_2 are not disjoint events, hence,

$$\mathbb{P}(A_1 \cup B_2) = \mathbb{P}(A_1) + \mathbb{P}(B_2) - \mathbb{P}(A_1 \cap B_2).$$

Further it holds that $\mathbb{P}(A_1 \cap B_2) = \mathbb{P}(A_1)\mathbb{P}(B_2)$. This follows from independence (further explored in Section 2.3). Now that we have identified the error, we can predict the resulting output.

$$\mathbb{P}(A_1 \cup B_2) = \mathbb{P}(A_1) + \mathbb{P}(B_2) - \mathbb{P}(A_1)\mathbb{P}(B_2) = \frac{5}{26} + \frac{3}{26} - \frac{5}{26} \cdot \frac{3}{26} \approx 0.2855.$$

It can be seen from the code output below, which repeats the comparison 5 times, that `mcEst1` consistently underestimates the desired probability, yielding estimates near 0.2855 instead.

<code>mcEst1</code>	<code>mcEst2</code>
0.285158	0.307668
0.285686	0.307815
0.285022	0.308132
0.285357	0.307261
0.285175	0.306606

- In lines 11-15 a `for` loop is implemented, which generates 5 MC predictions total. Note that lines 12 and 13 contain the main logic of this example.
- Line 12 is our incorrect simulation, and yields incorrect estimates. See the text above for a detailed explanation as to why the use of the `||` operator is incorrect in this case.
- Line 13 is our correct simulation, and for large N yields results close to the expected result. Note that the `union` function is used on `A` and `B`, instead of the “or” operator `||` used on line 12. The important point is that only a single sample is generated for each iteration of the composition.

Secretary with Envelopes

Now consider a more general form of the inclusion exclusion principle applied to a collection of sets, C_1, \dots, C_n . It is presented below, written in two slightly different forms:

$$\begin{aligned} \mathbb{P}\left(\bigcup_{i=1}^n C_i\right) &= \sum_{i=1}^n \mathbb{P}(C_i) - \sum_{\text{pairs}} \mathbb{P}(C_i \cap C_j) + \sum_{\text{triplets}} \mathbb{P}(C_i \cap C_j \cap C_k) - \dots + (-1)^{n-1} \mathbb{P}(C_1 \cap \dots \cap C_n) \\ &= \sum_{i=1}^n \mathbb{P}(C_i) - \sum_{i < j} \mathbb{P}(C_i \cap C_j) + \sum_{i < j < k} \mathbb{P}(C_i \cap C_j \cap C_k) - \dots + (-1)^{n-1} \mathbb{P}\left(\bigcap_{i=1}^n C_i\right). \end{aligned}$$

Notice that there are n major terms. These begin with the probabilities of individual events, then move onto pairs, continues with triplets, and follows these same lines until reaching a single final term involving a single intersection of all the sets. The ℓ 'th term has $\binom{n}{\ell}$ summands. For example,

there are $\binom{n}{2}$ pairs, $\binom{n}{3}$ triplets, etc. Notice also the alternating signs via $(-1)^{\ell-1}$. It is possible to conceptually see the validity of this formula for the case of $n = 3$ by drawing a *Venn diagram* and seeing the role of all summands.

Let us now consider a classic example that uses this inclusion exclusion principle. Assume that a secretary has an equal number of pre-labelled envelopes and letters, n . Suppose now that, at the end of the day, she is in such a rush to go home that she puts each letter in an envelope at random without any thought of matching the letter to its intended recipient on the envelope. What is then the probability of all letters being in wrong envelopes?

As an aid, let A_i be the event that the i 'th letter is put in the correct envelope. We have a handle on events involving intersections of distinct A_i values. For example, if $n = 10$, then

$$\mathbb{P}(A_1 \cap A_3 \cap A_7) = \frac{7!}{10!}.$$

Or more generally, the probability of an intersection of k such events is $p_k := (n - k)!/n!$.

The event we are seeking to evaluate is, $B = A_1^c \cap A_2^c \cap \dots \cap A_n^c$. Hence by *De Morgan's laws*,

$$B^c = A_1 \cup \dots \cup A_n.$$

Hence using the inclusion exclusion formula, using p_k and simplifying factorials and binomial coefficients,

$$\begin{aligned}\mathbb{P}(B) &= 1 - \mathbb{P}(A_1 \cup \dots \cup A_n) \\ &= 1 - \sum_{k=1}^n (-1)^{k+1} \binom{n}{k} p_k \\ &= 1 - \sum_{k=1}^n \frac{(-1)^{k+1}}{k!} \\ &= \sum_{k=0}^n \frac{(-1)^k}{k!}.\end{aligned}$$

Observe that as $n \rightarrow \infty$ this probability converges to $1/e \approx 0.3679$, yielding a simple *asymptotic approximation*. Listing 2.8 below evaluates $\mathbb{P}(B)$ in several alternative ways for $n = 1, 2, \dots, 8$. The function `bruteSetsProbabilityAllMiss()` works by creating all possibilities and counting. Although a highly inefficient way of evaluating $\mathbb{P}(B)$, it used here as it is instructive. The function `formulaCalcAllMiss()` evaluates the analytic solution from the formula derived above. Finally, the function `mcAllMiss()` estimates the probability via Monte Carlo simulation.

Listing 2.8: Secretary with envelopes

```

1  using Random, PyPlot, StatsBase, Combinatorics
2  Random.seed!(1)
3  function bruteSetsProbabilityAllMiss(n)
4      omega = collect(permutations(1:n))
5      matchEvents = []
6
7      for i in 1:n
8          event = []

```

```

9         for p in omega
10        if p[i] == i
11            push!(event,p)
12        end
13    end
14    push!(matchEvents,event)
15 end
16
17 noMatch = setdiff(omega,union(matchEvents...))
18 return length(noMatch)/length(omega)
19 end
20
21 function formulaCalcAllMiss(n)
22     return sum([(-1)^k/factorial(k) for k in 0:n])
23 end
24
25 function mcAllMiss(n,N)
26     function envelopeStuffer()
27         envelopes = Random.shuffle!(collect(1:n))
28         return sum([envelopes[i] == i for i in 1:n]) == 0
29     end
30
31     data = [envelopeStuffer() for _ in 1:N]
32     return sum(data)/N
33 end
34
35 N = 10^6
36
37 println("n\tBrute Force\tFormula\tMonte Carlo\tAnalytic",)
38 for n in 1:8
39     bruteForce = bruteSetsProbabilityAllMiss(n)
40     fromFormula = formulaCalcAllMiss(n)
41     fromMC = mcAllMiss(n,N)
42     println(n," \t",round(bruteForce,digits=4),"\t\t",round(fromFormula,digits=4),
43             "\t\t",round(fromMC,digits=4),"\t\t",round(1/MathConstants.e,digits=4))
44 end

```

n	Brute Force	Formula	Monte Carlo	Analytic
1	0.0	0.0	0.0	0.3679
2	0.5	0.5	0.4994	0.3679
3	0.3333	0.3333	0.3337	0.3679
4	0.375	0.375	0.3747	0.3679
5	0.3667	0.3667	0.3665	0.3679
6	0.3681	0.3681	0.3678	0.3679
7	0.3679	0.3679	0.3686	0.3679
8	0.3679	0.3679	0.3682	0.3679

- Lines 3- 19 define the function `bruteSetsProbabilityAllMiss()`, which uses a brute force approach to calculate $\mathbb{P}(B)$.
- Line 4 creates `omega`, a Set constructed via the `collect()` function.
- The nested loops in lines 7–15 populate the array `mathEvents` with elements of `omega` that have a match. The inner loop on lines 9–13, puts elements from `omega` in `event` if they satisfy an i 'th match. Then in line 16, `matchEvents[i]` describes the event A_i .

- In line 17, notice the use of the 3 dots *splat operator*, `....`. Here `union()` is applied to all the elements of `matchEvents`.
- then the return value in line 18 is a direct implementation via counting the elements of `noMatch`.
- Lines 21–23 implement the formulate derived above in straight forward manner.
- Lines 25–33 implement our function, `mcAllMiss()` that estimates the probability via Monte Carlo. The inner function, `envelopeStuffer()` returns a result from single experiment. Note the use of `shuffle!` in line 27, for creating a random permutation.

An Occupancy Problem

We now consider a problem related to the previous example. Imagine now the secretary placing r business cards randomly into n envelopes with $r \geq n$ and no limit on the number of business cards that can fit in an envelope. We now ask what is the probability that all envelopes are non-empty (i.e. occupied)?

To begin, denote A_i as the event that the i 'th envelope is empty, and hence A_i^c is the event that the i 'th envelope is occupied. Hence as before, we are seeking the probability of the event $B = A_1^c \cap A_2^c \cap \dots \cap A_n^c$. Using the same logic as in the previous example,

$$\begin{aligned}\mathbb{P}(B) &= 1 - \mathbb{P}(A_1 \cup \dots \cup A_n) \\ &= 1 - \sum_{k=1}^n (-1)^{k+1} \binom{n}{k} \tilde{p}_k,\end{aligned}$$

where \tilde{p}_k is the probability of at least k envelopes being empty. Now from basic counting considerations,

$$\tilde{p}_k = \frac{(n-k)^r}{n^r} = \left(1 - \frac{k}{n}\right)^r.$$

Thus we arrive at,

$$\mathbb{P}(B) = 1 - \sum_{k=1}^n (-1)^{k+1} \binom{n}{k} \left(1 - \frac{k}{n}\right)^r = \sum_{k=0}^n (-1)^k \binom{n}{k} \left(1 - \frac{k}{n}\right)^r.$$

We now calculate $\mathbb{P}(B)$ in Listing 2.9 below and compare them to Monte Carlo simulation estimates. In the code we consider several situations, by varying the number of envelopes from $n = 1, \dots, 100$. In addition, for every n , we let the number of business cards $r = Kn$ for $K = 2, 3, 4$.

Listing 2.9: Occupancy problem

```

1  using PyPlot
2
3  function occupancyAnalytic(n,r)
4      return sum([(-1)^k * binomial(n,k) * (1 - k/n)^r for k in 0:n])
5  end
6
7  function occupancyMC(n,r,N)
8      fullCount = 0

```

```

9      for _ in 1:N
10     envelopes = zeros(Int,n)
11     for k in 1:r
12       target = rand(1:n)
13       envelopes[target] += 1
14     end
15     numFilled = sum(envelopes .> 0)
16     if numFilled == n
17       fullCount += 1
18     end
19   end
20   return fullCount/N
21 end
22
23 max_n = 100
24 N = 10^3
25
26 dataAnalytic2 = [occupancyAnalytic(big(n),big(2*n)) for n in 1:max_n]
27 dataAnalytic3 = [occupancyAnalytic(big(n),big(3*n)) for n in 1:max_n]
28 dataAnalytic4 = [occupancyAnalytic(big(n),big(4*n)) for n in 1:max_n]
29
30 dataMC2 = [occupancyMC(n,2*n,N) for n in 1:max_n]
31 dataMC3 = [occupancyMC(n,3*n,N) for n in 1:max_n]
32 dataMC4 = [occupancyMC(n,4*n,N) for n in 1:max_n]
33
34 plot(1:max_n,dataAnalytic2,"b",label="K=2")
35 plot(1:max_n,dataAnalytic3,"r",label="K=3")
36 plot(1:max_n,dataAnalytic4,"g",label="K=4")
37 plot(1:max_n,dataMC2,"k+")
38 plot(1:max_n,dataMC3,"k+")
39 plot(1:max_n,dataMC4,"k+")
40 xlim(0,max_n)
41 ylim(0,1)
42 xlabel("n")
43 ylabel("Probability")
44 legend(loc="upper right");

```

- In lines 3-5 we create the function `occupancyAnalytic()`, which calculates the analytic solution to our occupancy problem using the formula derived above. It takes two arguments, the number of envelopes n and the number of business cards r . Note the use of the Julia function `binomial`.
- Lines 7-21 define the function `occupancyMC()`, which approximates $\mathbb{P}(B)$ for specific inputs via Monte Carlo simulation. Note the additional argument N , which is the total number of simulation runs.
- Line 8 defines the variable `fullcount`, which represents the total number of times all envelopes are full.
- Lines 9-19 contains the core logic of this function, and represent the act of the secretary assigning all the business cards randomly to the envelopes, and repeating this process N times total. Observe that in this `for` loop, there is no need to keep a count of the loop iteration number, hence for clarity we use underscore in line 9.
- Line 10 represents the number of business cards in each envelope before a single filing session.

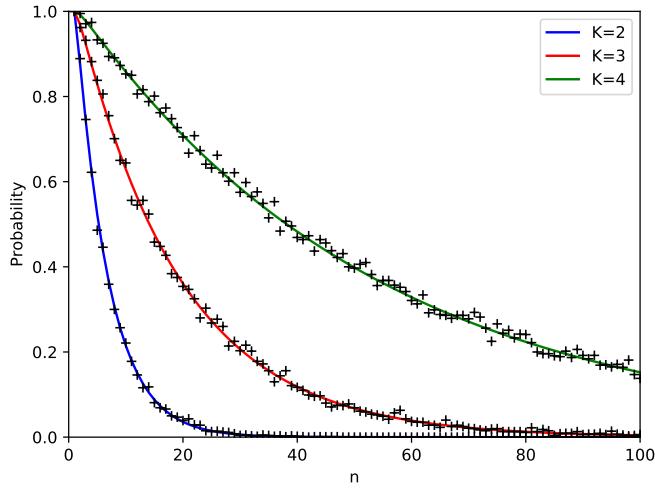


Figure 2.4: Analytic and estimated probabilities that no envelopes are empty, for various cases of n envelopes, and Kn business cards.

- Lines 11-14 represent the act of the secretary uniformly and randomly assigning all r business cards to one of the 1 to n envelopes.
- Line 15 then checks each element of `envelopes` to see if they are empty (i.e 0), and returns the total number of envelopes which are not empty. Note the use of element-wise comparison: `.>`, resulting in an array of boolean values that can be summed.
- Lines 16-18 then checks if all envelopes have been filled, and if so increments `fullCount` by 1.
- Line 20 then returns the proportion of times that all envelopes were filled.
- Lines 23-24 set the upper limit of the number of envelopes we will be simulating, and the number of simulation runs performed for each individual case N . For large N , the simulated results approximate that of the analytic solution.
- In lines 26-28 we calculate the analytic solution for various combinations of n envelopes and Kn business cards, for $K = 2, 3, 4$. Note the use of the `big` function to ensure that Julia compiles a method of `occupancyAnalytic()` suited for `big` values. Otherwise, the `binomial` will overflow.
- In lines 30-32 we calculate numerical approximations of the same cases calculated in lines 26-28. Note that in each specific case, N simulations are run.
- Lines 34-44 are used for plotting Figure 2.4. Note that the Monte Carlo estimates are plotted in black, using the '+' symbol.

2.3 Independence

We now look at *independence* and *independent events*. Two events, A and B , are said to be independent if the probability of their *intersection* is the product of their probabilities: $\mathbb{P}(AB) = \mathbb{P}(A)\mathbb{P}(B)$. A classic example is a situation where the random experiment involves physical components that are assumed to not interact, for example flipping two coins. Independence is often a modeling assumption and plays a key role in most of the statistical models presented in the remainder of the book.

To explore independence, it is easiest to consider a situation where it does not hold. Consider drawing a number uniformly from the range $10, 11, \dots, 25$. What is the probability of getting the number 13? Clearly there are $25 - 10 + 1 = 16$ options, and hence the probability is $1/16 = 0.0625$. However, the event of obtaining 13 could be described as the intersection of the events $A := \{\text{first digit is } 1\}$ and $B := \{\text{second digit is } 3\}$. The probabilities of which are $10/16 = 0.625$ and $2/16 = 0.125$ respectively. Notice that the product of these probabilities is not 0.0625, but rather $20/256 = 0.078125$. Hence we see that, $\mathbb{P}(AB) \neq \mathbb{P}(A)\mathbb{P}(B)$ and the events are not independent.

One way of viewing this lack of independence is as follows. Witnessing the event A gives us some information about the likelihood of B . Since if A occurs then we know that the number is in the range $10, \dots, 19$ and hence there is a $1/10$ chance for B to occur. However, if A does not occur then we lie in the range $20, \dots, 25$ and there is a $1/6$ chance for B to occur.

If however we change the range of random digits to be $10, \dots, 29$ then the two events are independent. This can be demonstrated by running Listing 2.10 below, and then modifying line 1.

Listing 2.10: Independent events

```

1  using Random
2  Random.seed!(1)
3
4  numbers = 10:25
5  N = 10^7
6
7  firstDigit(x) = Int(floor(x/10))
8  secondDigit(x) = x%10
9
10 numThirteen, numFirstIsOne, numSecondIsThree = 0, 0, 0
11
12 for _ in 1:N
13     X = rand(numbers)
14     numThirteen += X == 13 ? 1 : 0
15     numFirstIsOne += firstDigit(X) == 1 ? 1 : 0
16     numSecondIsThree += secondDigit(X) == 3 ? 1 : 0
17 end
18
19 probThirteen, probFirstIsOne, probSecondIsThree =
20     (numThirteen, numFirstIsOne, numSecondIsThree). / N
21
22 println("P(13) = ", round(probThirteen, digits=4),
23         "\nP(1_) = ", round(probFirstIsOne, digits=4),
24         "\nP(_3) = ", round(probSecondIsThree, digits=4),
25         "\nP(1_)*P(_3) = ", round(probFirstIsOne*probSecondIsThree, digits=4))
```

```

P(13) = 0.0626
P(1_) = 0.6249
P(_3) = 0.1252
P(1_)*P(_3) = 0.0783

```

- Lines 4 and 5 set the range of digits considered and number of simulation runs respectively.
- Line 7 defines a function which returns the first digit of our number through the use of the `floor` function, and converting the resulting value to an integer type.
- Line 8 defines a function, which uses the *modulus* operator `%` to return the second digit of our number.
- In line 10 we initialize the three placeholder variables, which represent the number chosen, and its first and second digits respectively.
- Lines 12-17 contains the core logic of this example where we use a Monte Carlo. After generating a random number (line 13), on each of the lines 14, 15 and 16 we increment the count by 1 in case the specified condition is met.
- Line 19-20 evaluates the total proportions.

2.4 Conditional Probability

It is often the case that knowing an event has occurred, say B , modifies our belief about the chances of another event occurring, say A . This concept is captured via the *conditional probability* of A given B , denoted by $\mathbb{P}(A | B)$ and defined for B where $\mathbb{P}(B) > 0$. In practice, given a probability model, $\mathbb{P}(\cdot)$ we construct the conditional probability, $\mathbb{P}(\cdot | B)$ via,

$$\mathbb{P}(A | B) := \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}. \quad (2.4)$$

As an elementary example, refer back to Table 2.1 depicting the outcome of rolling two dice. Let now B be the event of the sum being greater than or equal to 10, in other words,

$$B = \{(i, j) \mid i + j \geq 10\}.$$

To help illustrate this further, consider a game player who rolls the dice without showing us the result, and then poses to us the following: “The sum is greater or equal to 10. Is it even or odd?”. Let A be the event of the sum being even. We then evaluate,

$$\begin{aligned} \mathbb{P}(A | B) &= \mathbb{P}(\text{Sum is even} | B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)} = \frac{\mathbb{P}(\text{Sum is 10 or 12})}{\mathbb{P}(\text{Sum is } \geq 10)} = \frac{4/36}{6/36} = \frac{2}{3}, \\ \mathbb{P}(A^c | B) &= \mathbb{P}(\text{Sum is odd} | B) = \frac{\mathbb{P}(A^c \cap B)}{\mathbb{P}(B)} = \frac{\mathbb{P}(\text{Sum is 11})}{\mathbb{P}(\text{Sum is } \geq 10)} = \frac{2/36}{6/36} = \frac{1}{3}. \end{aligned}$$

It can be seen that given B , it is more likely that A occurs (even) as opposed to A^c (odd), hence we are better off answering “even”.

The Law of Total Probability

Often our probability model is comprised of conditional probabilities as elementary building blocks. In such cases, equation (2.4) may be better viewed as,

$$\mathbb{P}(A \cap B) = \mathbb{P}(B) \mathbb{P}(A | B).$$

This is particularly useful when there exists some *partition* of Ω , namely, $\{B_1, B_2, \dots\}$. A partition of a set U is a collection of non-empty sets that are mutually disjoint and whose union is U . Such a partition allows us to represent A as a disjoint union of the sets $A \cap B_k$, and treat $\mathbb{P}(A | B_k)$ as model data. In such a case, we have the *law of total probability*

$$\mathbb{P}(A) = \sum_{k=0}^{\infty} \mathbb{P}(A \cap B_k) = \sum_{k=0}^{\infty} \mathbb{P}(A | B_k) \mathbb{P}(B_k).$$

As an exotic fictional example, consider the world of semi-conductor manufacturing. Room cleanliness in the manufacturing process is critical, and dust particles are kept to a minimum. Let A be the event of a manufacturing failure, and assume that it depends on the number of dust particles via,

$$\mathbb{P}(A | B_k) = 1 - \frac{1}{k+1},$$

where B_k is the event of having k dust particles in the room ($k = 0, 1, 2, \dots$). Clearly the larger k , the higher the chance of manufacturing failure. Assume further that

$$\mathbb{P}(B_k) = \frac{6}{\pi^2(k+1)^2} \quad \text{for } k = 0, 1, \dots$$

From the well known *Basel Problem*, we have $\sum_{k=1}^{\infty} k^{-2} = \pi^2/6$. This implies that $\sum \mathbb{P}(B_k) = 1$.

Now we ask, what is the probability of manufacturing failure? The analytic solution is given by,

$$\mathbb{P}(A) = \sum_{k=0}^{\infty} \mathbb{P}(A | B_k) \mathbb{P}(B_k) = \sum_{k=0}^{\infty} \left(1 - \frac{1}{k+1}\right) \frac{6}{\pi^2(k+1)^2}.$$

This infinite series, can be explicitly evaluated at,

$$\mathbb{P}(A) = \frac{\pi^2 - 6\zeta(3)}{\pi^2},$$

where $\zeta(\cdot)$ is the *Riemann Zeta Function*,

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}.$$

The code Listing 2.11 below approximates the infinite series numerically (truncating at $n = 2000$) and compares to the analytic solution. simulation to approximate the probability of manufacturing failure, and compares this estimate against the analytically expected result.

Listing 2.11: Defects in manufacturing

```
1  using Random, SpecialFunctions
```

```

2 Random.seed!(1)
3
4 n = 2000
5
6 probAgivenB(k) = 1 - 1/(k+1)
7 probB(k) = 6/(pi*(k+1))^2
8
9 numerical= sum([ probAgivenB(k)*probB(k) for k in 0:n])
10
11 analytic = (pi^2 - 6*zeta(3))/pi^2
12
13 analytic, numerical

```

(0.2692370305985609, 0.26893337073278945)

- This listing is self-explanatory, however note the use of the Julia function `zeta()` from the `SpecialFunctions` package in line 11.

2.5 Bayes' Rule

Baye's rule (or *Bayes' theorem*) is nothing but a simple manipulation of (2.4) yielding,

$$\mathbb{P}(A | B) = \frac{\mathbb{P}(B | A)\mathbb{P}(A)}{\mathbb{P}(B)}. \quad (2.5)$$

However, the consequences are far reaching. Often we observe a *posterior outcome* or measurement, say the event B , and wish to evaluate the probability of a *prior condition*, say the event A . That is, given some measurement or knowledge we wish to evaluate how likely is it that a prior condition occurred. Equation (2.5) allows us to do just that.

Was it a 0 or a 1?

As an example, consider a communication channel involving a stream of transmitted bits (0's and 1's), where 70% of the bits are 1, and the rest 0. A typical snippet from the channel ...0101101011101111101....

The channel is imperfect due to physical disturbances (e.g. interfering radio signals) and that bits received are sometimes distorted. Hence there is a chance (ϵ_0) of interpreting a bit as 1 when it is actually 0, and similarly, there is a chance (ϵ_1) of interpreting a bit as 0 when it is actually 1.

Now say that we received (Rx) a bit, and interpreted it as 1 (this is the posterior outcome). What is the chance that it was in-fact transmitted (Tx) as a 1? Applying Bayes' rule:

$$\mathbb{P}(\text{Tx 1} | \text{Rx 1}) = \frac{\mathbb{P}(\text{Rx 1} | \text{Tx 1})\mathbb{P}(\text{Tx 1})}{\mathbb{P}(\text{Rx 1})} = \frac{(1 - \epsilon_1)0.7}{0.7(1 - \epsilon_1) + 0.3\epsilon_0}.$$

For example, if $\epsilon_0 = 0.1$ and $\epsilon_1 = 0.05$ we have that $\mathbb{P}(\text{Tx 1} | \text{Rx 1}) = 0.9568$. The code in Listing 2.12 below illustrates this via simulation.

Listing 2.12: Tx Rx Bayes

```

1  using Random
2  Random.seed!(1)
3
4  N = 10^5
5  prob1 = 0.7
6  eps0 = 0.1
7  eps1 = 0.05
8
9  function flipWithProb(bit,prob)
10    return rand() < prob ? xor(bit,1) : bit
11  end
12
13  TxDATA = rand(N) .< prob1
14  RXDATA = [x == 0 ? flipWithProb(x,eps0) : flipWithProb(x,eps1) for x in TxDATA]
15
16  numRx1 = 0
17  totalRx1 = 0
18  for i in 1:N
19    if RXDATA[i] == 1
20      totalRx1 += 1
21      numRx1 += TXDATA[i]
22    end
23  end
24
25  numRx1/totalRx1, ((1-eps1)*0.7)/((1-eps1)*0.7+0.3*eps0)

```

(0.9576048007598325, 0.9568345323741007)

- In lines 9 to 11 the function `flipWithProb()` is defined. It uses the `xor` function to randomly flip the input argument `bit`, according to the rate given by the argument `prob`. Note that a short-handed version of an `if` statement is used here. That is, the expression before `?` is checked if true, and if it is `xor()` is evaluated, else the original argument `bit` is returned.
- Line 13 generates the array `TxDATA`, which contains true and false values representing our transmitted bits of 1's and 0's respectively. It does this by uniformly and randomly generating numbers on the range $[0, 1]$, and then evaluating element-wise if they are less than the specified probability of receiving a 1, `prob`.
- Line 14 generates the array `RXDATA`, which represents our simulated received data. First the type of received bit is checked, and the `flipWithProb()` function is used to flip received bits at the rate specified in lines 6 and 7, depending on if the received bit is a 0 or 1.
- In lines 16 and 17 the counters `numRx1` and `totalRx1` are initialized. These represent the total number of correctly transmitted and received 1's and total number of transmitted 1's respectively.
- Lines 18-23 are used to check the nature of all bits. If the bit received is 1, then it increments the counter `totalRx1` by 1. It also increments the counter `numRx1` by the value of the transmitted bit (which may be 1, but could also be 0).
- In line 25 the proportion of times that a 1 was transmitted and correctly registered as a 1 is calculated. This value is compared against the analytic solution from the equation above.

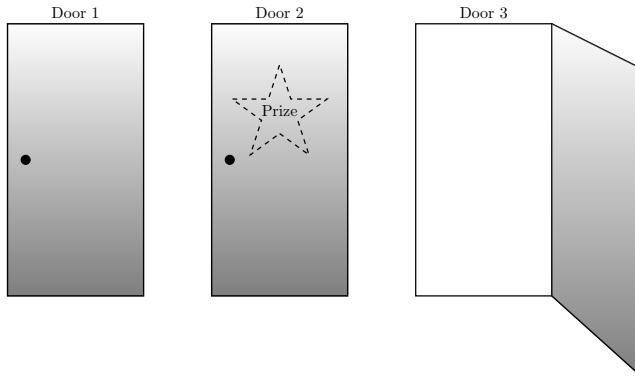


Figure 2.5: Monty Hall: If the prize is behind Door 2 and Door 1 is chosen, the game show host must reveal door 3.

The Monty Hall Problem

The *Monty Hall problem* is a famous problem which was first posed and solved in 1975 by the mathematician Steve Selvin [SBK75]. It is a famous example illustrating how probabilistic reasoning may sometimes yield to surprising results.

Consider a contestant on a television game show, with three doors in front of her. One of the doors contains a prize, while the other two are empty. The contestant is then asked to guess which door contains the prize, and she makes a random guess. Following this the game show host (GSH) reveals an empty (losing) door from one of the two remaining doors not chosen. The contestant is then asked if she wishes to stay with their original choice, or if she wishes to switch to the remaining closed door. Following the choice of the contestant to stay or switch, the door with the prize is revealed. The question is: should the contestant stay with their original choice, or switch? Alternatively, perhaps it doesn't matter.

For example, in Figure 2.5 we see the situation where the hidden prize is behind door 2. Say the contestant has chosen door 1. In this case, the GSH has no choice but to reveal door 3. Alternatively, if the contestant has chosen door 2, then the GSH will reveal either door 1 or door 3.

The two possible policies for the contestant are:

Policy I - Stay with their original choice after the door is revealed.

Policy II- Switch after the door is revealed.

Let us consider the probability of winning for the two different policies. If the player adopts Policy I then she always stays with her initial guess regardless of the GSH action. In this case her chance of success is $1/3$; that is, she wins if her initial choice is correct.

However if she adopts Policy II then she always switches after the GSH reveals an empty room. In this case we can show that her chance of success is $2/3$; that is, she actually wins if her initial guess is incorrect. This is because the GSH must always reveal a losing door. If she originally chose

a losing door, then he must reveal the second losing door every time (otherwise he would reveal the prize). That is, if she chooses an incorrect door at the start, the non-revealed door will always be the winning door. The chance of such an event is $2/3$.

As a further aid for understanding imagine a case of 100 doors and a single prize behind one of them. In this case assume that the player chooses a door (for example door 1), and following this the GSH reveals 98 losing doors. There are now only two doors remaining, her choice door 1, and (say for example), door 38. The intuition of the problem suddenly becomes obvious. The player's original guess was random and hence door 1 had a $1/100$ chance of containing the prize, however the GSH's actions were constrained. He had to reveal only losing doors, and hence there is a $99/100$ chance that door 38 contains the prize. Hence Policy II is clearly superior.

Back to the case of 3 doors. We now analyze it by applying Bayes' theorem. Let A_i be the event that the prize is behind door i . Let B_i be the event that door i is revealed by the GSH. Then if for example the player initially chooses door 1 and then the GSH reveals door 2, we have the following:

$$\begin{aligned}\mathbb{P}(A_1 | B_2) &= \frac{\mathbb{P}(B_2 | A_1)\mathbb{P}(A_1)}{\mathbb{P}(B_2)} = \frac{\frac{1}{2} \times \frac{1}{3}}{\frac{1}{2}} = \frac{1}{3}, && \text{(Policy I)} \\ \mathbb{P}(A_3 | B_2) &= \frac{\mathbb{P}(B_2 | A_3)\mathbb{P}(A_3)}{\mathbb{P}(B_2)} = \frac{\frac{1}{2} \times \frac{1}{3}}{\frac{1}{2}} = \frac{2}{3}. && \text{(Policy II)}\end{aligned}$$

In the second case note that $\mathbb{P}(B_2 | A_3) = 1$ because the GSH must reveal door 2 if the prize is behind door 3 since door 1 was already picked. Hence we see that while neither policy guarantees a win, Policy II clearly dominates Policy I.

Now that we have shown this analytically, we perform a Monte Carlo simulation of the Monty Hall problem in Listing 2.13 below.

Listing 2.13: The Monty Hall problem

```

1  using Random
2  Random.seed!(1)
3
4  function montyHall(switchPolicy)
5      prize = rand(1:3)
6      choice = rand(1:3)
7
8      if prize == choice
9          revealed = rand(setdiff(1:3,choice))
10     else
11         revealed = rand(setdiff(1:3,[prize,choice]))
12     end
13
14     if switchPolicy
15         choice = setdiff(1:3,[revealed,choice])[1]
16     end
17
18     return choice == prize

```

```

19 end
20
21 N = 10^6
22 sum([montyHall(true) for _ in 1:N])/N,
23 sum([montyHall(false) for _ in 1:N])/N

```

(0.667087, 0.332973)

- In lines 4 to 19 the function `montyHall()` is defined, which performs one simulation run of the problem given a particular policy, or strategy of play. The two possible strategies of the player is to switch her initial choice, or stay with her initial choice, after the revelation of a losing door by the GSH. The policy is set by `true` (switch, i.e. Policy II) or `false` (stay, i.e. Policy I) as an argument. We now explain the workings of this function in more detail.
- Line 5 represents which door contains the prize. It is uniformly and randomly selected, and stored as the variable `prize`.
- Line 6 represents our initial first choice of door. It is also uniformly and randomly selected, and stored as the variable `choice`.
- Lines 8-12 contain the logic and action of the GSH. Since he knows the location of both the prize and the chosen door, he first mentally checks if they are the same. If they are he reveals a door according to line 6, if not, then he process to reveal a door according to the logic in line 8. In either case the revealed door is stored by the variable `revealed`.
- Line 9 represents his action if the initial `choice` door is the same as the `prize` door. In this case he is free to reveal either of the remaining two doors, i.e. the set difference between all doors, and the player's `choice` door. Note that in this case the set difference has 2 elements.
- Line 11 represents the GSH action if the `choice` door is different to the `prize` door. In this case his hand is forced, as he cannot reveal the player's chosen door or the `prize` door, he is forced to reveal the one remaining door, which can be thought of as the set difference between `1:3` (all doors) and `[prize, choice]`. Note that in this case the set difference has a single element.
- Line 14 represents the player's action, after the GSH revelation, based on either a switch (`true`) or stay (`false`) policy. If the contestant chooses to stay with her initial guess (`false`), then we skip to Line 18. However, if she chooses to swap (`true`), then we reassign our initial `choice` to the one remaining door. This is done on line 15. Note the use of `[1]`, which is used assign the value of the array to `choice`, rater than the array itself.
- Line 18 checks if the player's `choice` is the same as the `prize`, and returns `true` if the she wins, and `false` if she loses.
- Lines 22 and 23 produce Monte Carlo estimates. It can be observed that the swap strategy (Policy II) results in winning about 66% of the time, whereas the stay strategy (Policy I) wins only 33% of the time.

Chapter 3

Probability Distributions - DRAFT

In this chapter, we introduce random variables, different types of distributions and related concepts. In the previous chapter we explored probability spaces without much emphasis on numerical random values. However, when carrying out random experiments, there are almost always numerical values involved. In the context of probability, these values are often called *random variables*. Mathematically, a random variable X is a function of the sample space, Ω , and takes on integer, real, complex, or even a vector of values. That is, for every possible outcome $\omega \in \Omega$, there is some possible outcome, $X(\omega)$.

The chapter is organized as follows: In Section 3.1 we introduce the concept of a random variable and its probability distribution. In Section 3.2 we introduce the mean, variance and other numerical descriptors of probability distributions. In Section 3.3 we explore several alternative functions for describing probability distributions. In Section 3.4 we focus on the Julia's interface for probability distributions, namely the `distributions` package. Then Section 3.5 explores a variety of discrete distributions. This is followed by Section 3.6 where we explore some continuous distributions together with additional concepts such as hazard rates and more. We close with Section 3.7, where we explore multi-dimensional probability distributions.

3.1 Random Variables

As an example, let us consider Ω which consists of 6 names. Assume that the probability function, \mathbb{P} , assigns uniform probabilities to each of the names. Let now, $X : \Omega \rightarrow \mathbb{Z}$, be the function (i.e. random variable) that counts the number of letters in each name. The question is then finding:

$$p(x) := \mathbb{P}(X = x), \quad \text{for } k \in \mathbb{Z}.$$

The function $p(x)$ represents the *probability distribution* of the random variable X . In this case, since X measures name lengths, X is a discrete random variable, and its probability distribution may be represented by a *Probability Mass Function (PMF)*, such as $p(x)$.

To illustrate this, we carry out a simulation of many such random experiments, yielding many replications of the random variable X , which we then use to estimate $p(x)$. This is performed in

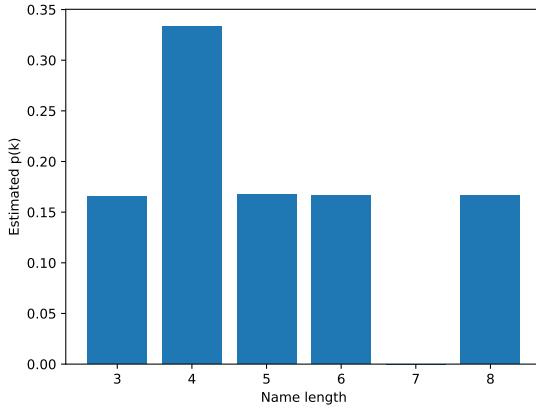


Figure 3.1: A discrete probability distribution taking values on $\{3, 4, 5, 6, 8\}$.

Listing 3.1 below.

Listing 3.1: A simple random variable

```

1  using StatsBase, PyPlot
2
3  names = ["Mary", "Mel", "David", "John", "Kayley", "Anderson"];
4  randomName() = rand(names)
5  X = 3:8
6  N = 10^6
7
8  sampleLengths = [length(randomName()) for _ in 1:N]
9  bar(X, counts(sampleLengths)/N);
10 xlabel("Name length")
11 ylabel("Estimated p(k)")

```

- In line 3 we create the array `names`, which contains names with different character lengths. Note that two names have four characters “Mary” and “John”, while there is no name with 7 characters.
- In line 4, we define the function `randomNames()` which randomly selects, with equal probability, an element from the array `names`.
- In line 5, we specify that we will count names of 3 to 8 characters in length.
- Line 6 specifies how many random experiments of choosing a name we will perform.
- Line 8 uses a comprehension and the function `length()` to count the length of each random name, and stores the results in the array `sampleLengths`. Here the Julia function `length()` is the analog of the random variable. That is, it is a function of the sample space, Ω , yielding a numerical value.
- Line 9 uses the function `counts()` to count how many words are of length 3, 4, up to 8. The `bar()` function is then used to plot a bar-chart of the proportion of counts for each word length. Two key observations can be made. It can be seen that words of length 4 occurred twice as much as words of lengths 3, 5, 6 and 8. In addition, no words of length 7 were selected, as no name in our original array had a length of 7.

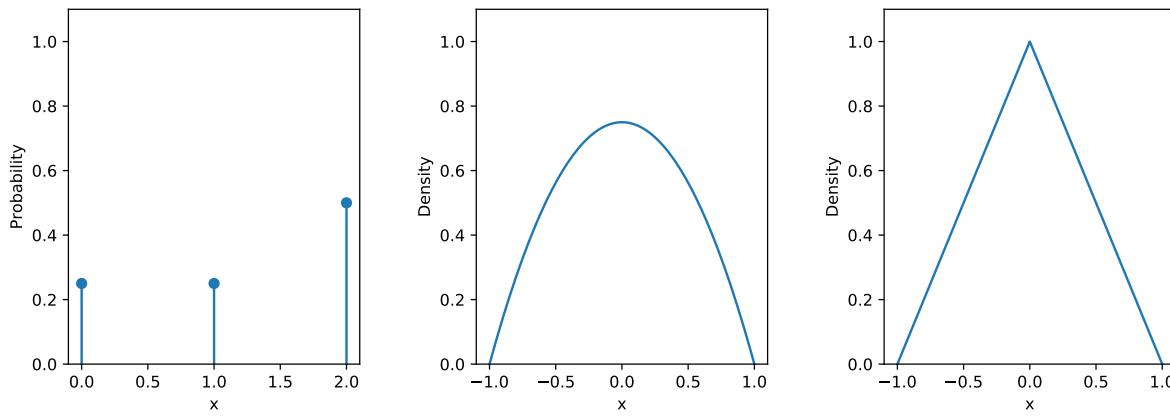


Figure 3.2: Three different examples of probability distributions.

Types of Random Variables

In the previous example, the random variable X took on discrete values and is thus called a *discrete random variable*. However, quantities measured in nature are often continuous, in which case a *continuous random variable* better describes the situation. For example, consider measuring the weights of people randomly selected from a big population.

In describing the probability distribution of a continuous random variable, the probability mass function, $p(x)$, as used above, is no longer applicable. This is because for a continuous random variable X , $\mathbb{P}(X = x)$ for any particular value of x is 0. Hence in this case, the *Probability Density Function (PDF)*, $f(x)$ is used, where,

$$f(x)\Delta \approx \mathbb{P}(x \leq X \leq x + \Delta).$$

Here the approximation becomes exact as $\Delta \rightarrow 0$. Figure 3.2, illustrates three examples of probability distributions. The one on the left is discrete and the other two are continuous.

The discrete probability distribution appearing on the left in Figure 3.2 can be represented mathematically by the probability mass function

$$p(x) = \begin{cases} 0.25 & \text{if } x = 0, \\ 0.25 & \text{if } x = 1, \\ 0.5 & \text{if } x = 2. \end{cases} \quad (3.1)$$

The smooth continuous probability distribution is defined by the probability density function

$$f_1(x) = \frac{3}{4}(1 - x^2) \quad \text{for } -1 \leq x \leq 1.$$

Finally, the triangular probability distribution is defined by the probability density function,

$$f_2(x) = \begin{cases} x + 1 & \text{if } x \in [-1, 0], \\ 1 - x & \text{if } x \in (0, 1]. \end{cases}$$

Note that for both the probability mass function and the probability density functions, it is implicitly assumed that $p(x)$ and $f(x)$ are zero for x values not specified in the equation.

It can be verified that for the discrete distribution,

$$\sum_x p(x) = 1,$$

and for the continuous distributions,

$$\int_{-\infty}^{\infty} f_i(x) dx = 1 \quad \text{for } i = 1, 2.$$

There are additional descriptors of probability distributions other than the PMF and PDF, and these are further discussed in Section 3.3. Note that Figure 3.2 was generated by Listing 3.2 below.

Listing 3.2: Plotting discrete and continuous distributions

```

1  using PyPlot
2
3  pDiscrete = [0.25, 0.25, 0.5]
4  xGridD = 0:2
5
6  pContinuous(x) = 3/4*(1 - x^2)
7  xGridC = -1:0.01:1
8
9  pContinuous2(x) = x < 0 ? x+1 : 1-x
10
11 figure(figsize=(12.4,4))
12 subplots_adjust(wspace=0.4)
13
14 subplot(131)
15 stem(xGridD,pDiscrete,basefmt="none")
16 ylim(0,1.1)
17 xlabel("x")
18 ylabel("Probability")
19
20 subplot(132)
21 plot(xGridC,pContinuous.(xGridC))
22 ylim(0,1.1)
23 xlabel("x")
24 ylabel("Density")
25
26 subplot(133)
27 plot(xGridC,pContinuous2.(xGridC))
28 ylim(0,1.1)
29 xlabel("x")
30 ylabel("Density");

```

- In lines 3 we define an array specifying the PMF of our discrete distribution, and in lines 6 and 9 we define functions specifying the PDFs of our continuous distributions.
- In lines 11-30 we create plots of each of our distributions. Note that in the discrete case we use the `stem()` function to plot the PMF, while in the continuous cases we use the `plot()` function.

3.2 Moment Based Descriptors

The probability distribution of a random variable fully describes the probabilities of the events such as $\{\omega \in \Omega : X(\omega) \in A\}$ for all sensible $A \subset \mathbb{R}$. However, it is often useful to describe the nature of a random variable via a single number or a few numbers. The most common example of this is the *mean* which describes the center of mass of the probability distribution. Other examples include the *variance* and *moments* of the probability distribution. We expand on these now.

Mean

The mean, also known as the *expected value* of a random variable X , is a measure of the central tendency of the distribution of X . It is represented by $\mathbb{E}[X]$, and is the value we expect to obtain “on average” if we continue to take observations of X and average out the results. The mean of a discrete distribution with PMF $p(x)$ is

$$\mathbb{E}[X] = \sum_x x p(x).$$

In the example of the discrete distribution given by (3.1) it is,

$$\mathbb{E}[X] = 0 \times 0.25 + 1 \times 0.25 + 2 \times 0.5 = 1.25.$$

The mean of a continuous random variable, with PDF $f(x)$ is

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x f(x) dx,$$

which in the examples of $f_1(\cdot)$ and $f_2(\cdot)$ from Section 3.1 yield,

$$\int_{-1}^1 x \frac{3}{4}(1-x^2) dx = 0,$$

and,

$$\int_{-1}^0 x+1 dx + \int_0^1 1-x dx = 0,$$

respectively. As can be seen, both distributions have the same mean even though their shapes are different. For illustration purposes, we now carry out this integration numerically in Listing 3.3 below.

Listing 3.3: Expectation via numerical integration

```

1  using QuadGK
2
3  sup = (-1,1)
4  f1(x) = 3/4*(1-x^2)
5  f2(x) = x < 0 ? x+1 : 1-x
6
7  expect(f,support) = quadgk((x) -> x*f(x),support[1],support[2])[1]
8
9  expect(f1,sup),expect(f2,sup)

```

(0.0, -2.0816681711721685e-17)

- In line 1 we call the QuadGK package, which contains functions that support one-dimensional numerical integration via a numerical integration method called *adaptive Gauss-Kronrod quadrature*.
- In lines 4 and 5 we define the PDF's of our functions functions `f1()` and `f2()` respectively.
- In line 7 we define the function `expect()` which takes two arguments, a function to integrate `f`, and a domain over which to integrate the function `support`. It uses the `quadgk` function to evaluate the 1-dimensional integral given above. Note that the start and end points of the integral are `support[1]` and `support[2]` respectively. Note also that the function `quadgk()` returns two arguments, the evaluated integral and an estimated upper bound on the absolute error. Hence [1] is included at the end of the function, so that only the integral is returned.
- Line 9 then evaluates the numerical integral of the functions `f1` and `f2` over the interval `sup`. As can be seen, both integrals are effectively evaluated to zero.

General Expectation and Moments

In general, for a function $h : \mathbb{R} \rightarrow \mathbb{R}$ and a random variable X , we can consider the random variable $Y := h(X)$. The distribution of Y will typically be different from the distribution of X . As for the mean of Y , we have,

$$\mathbb{E}[Y] = \mathbb{E}[h(X)] = \begin{cases} \sum_x h(x) p(x) & \text{for discrete,} \\ \int_{-\infty}^{\infty} h(x) f(x) dx & \text{for continuous.} \end{cases} \quad (3.2)$$

Note that the above expression does not require explicit knowledge of the distribution of Y but rather uses the distribution (PMF or PDF) of X .

A common case, is $h(x) = x^\ell$, in which case we call $\mathbb{E}[X^\ell]$, the ℓ 'th moment of X . Then, for a random variable X with PDF $f(x)$, the ℓ^{th} moment of X is,

$$\mathbb{E}[X^\ell] = \int_{-\infty}^{\infty} x^\ell f(x) dx.$$

Note that the first moment is the mean and the zero'th moment is always 1. The second moment, is related to the variance as we explain below.

Variance

The *variance* of a random variable X , often denoted $\text{Var}(X)$ or σ^2 , is a measure of the spread, or *dispersion*, of the distribution of X . It is defined by,

$$\text{Var}(X) := \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2. \quad (3.3)$$

Here we apply, (3.2) by considering $h(x) = (x - \mathbb{E}[X])^2$. The second formula, illustrating the role of the second moment, follows from expansion.

For the discrete distribution, (3.1), we have:

$$\text{Var}(X) = (0 - 1.25)^2 \times 0.25 + (1 - 1.25)^2 \times 0.25 + (2 - 1.25)^2 \times 0.5 = 0.6875.$$

For the continuous distributions from Section 3.1, $f_1(\cdot)$ and $f_2(\cdot)$, with respective random variables X_1 and X_2 , we have

$$\begin{aligned}\text{Var}(X_1) &= \int_{-1}^1 x^2 \frac{3}{4}(1-x^2) dx - (\mathbb{E}[X_1])^2 = \frac{3}{4} \left[\frac{x^3}{3} - \frac{x^5}{5} \right]_{-1}^1 - 0 = 0.2, \\ \text{Var}(X_2) &= \int_{-1}^0 x^2(x+1) dx + \int_0^1 x^2(1-x) dx - (\mathbb{E}[X_2])^2 = \frac{1}{6}.\end{aligned}$$

The variance of X can also be considered as the expectation of a new random variable, $Y := (X - \mathbb{E}[X])^2$. However, when considering variance, the distribution of Y is seldom mentioned. Nevertheless, as an exercise we explore this now. Consider a random variable X , with density,

$$f(x) = \begin{cases} x-4 & \text{if } x \in [4, 5], \\ 6-x & \text{if } x \in (5, 6]. \end{cases}$$

This density is similar to $f_2(\cdot)$ previously covered, but with support $[4, 6]$. In Listing 3.4, we generate random observations from X , and calculate data-points for Y based on these observations. We then plot both the distribution of X and Y , and show that the sample mean of Y is the sample variance of X . Note that our code uses some elements from the `Distributions` package, more of which is covered in Section 3.4.

Listing 3.4: Variance of X as a mean of Y

```

1  using Distributions, PyPlot
2  dist = TriangularDist(4, 6, 5)
3  N = 10^6
4  data = rand(dist, N)
5  yData=(data.-5).^2
6
7  figure(figsize=(10,5))
8  subplots_adjust(wspace=0.4)
9
10 subplot(121)
11 plt[:hist](data,100, normed="true")
12 xlabel("x")
13 ylabel("Proportion")
14
15 subplot(122)
16 plt[:hist](yData,100, normed="true")
17 xlabel("y")
18 ylabel("Proportion")
19
20 mean(yData), var(data)

```

(0.16654029017290403, 0.16654025029682298)

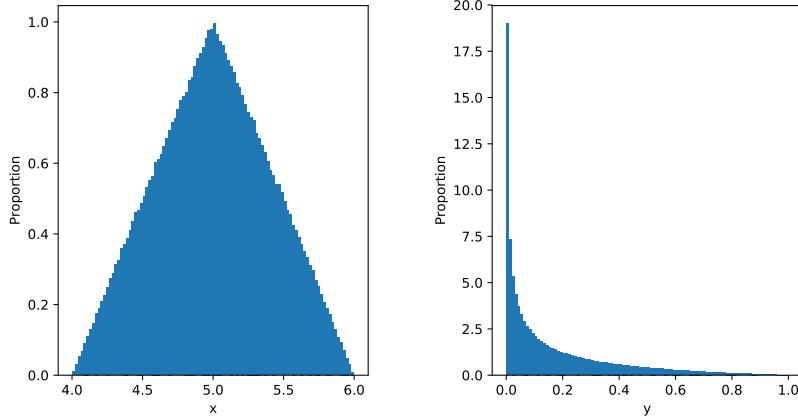


Figure 3.3: Histograms for samples of the random variables X and Y .

- Line 1 calls the `Distributions` package. This package supports a variety of distribution types through the many functions it contains. We expand further on the use of the `Distributions` package in Section 3.4.
- Line 2 uses the `Triangular()` function from the `Distributions` package to create a triangular distribution type object with a mean of 5 and symmetric shape over the bound [4, 6]. We assign this as the variable `dist`.
- In line 4 we generate an array of N observations from the distribution by applying the `rand()` function on the distribution `dist`.
- Line 5 takes the observations in `data` and from them generates observations for the new random variable Y . The values are stored in the array `yData`.
- Lines 7-15 are used to plot histograms of the data in the arrays `data` and `yData` via `plt[:hist]`. It can be observed that the histogram on the left approximates the PDF of our triangular distribution. The histogram on the right approximates the distribution of the new variable Y . This distribution is seldom considered when evaluating the variance of Y .
- Line 20 uses the functions `mean()` and `var()` on the arrays `yData` and `data` respectively. It can be seen from the output that the mean of the distribution Y is the same as the variance of X .

Higher Order Descriptors: Skewness and Kurtosis

As described above, the second moment plays a role defining the dispersion of a distribution via the variance. How about higher order moments? We now briefly define the skewness and kurtosis of a distribution utilizing the first three moments and first four moments respectively.

Take a random variable X with $\mathbb{E}[X] = \mu$ and $\text{Var}(X) = \sigma^2$, then the *skewness*, is defined as,

$$\gamma_3 = \mathbb{E}\left[\left(\frac{X - \mu}{\sigma}\right)^3\right] = \frac{\mathbb{E}[X^3] - 3\mu\sigma^2 - \mu^3}{\sigma^3},$$

and the *kurtosis* is defined as,

$$\gamma_4 = \mathbb{E}\left[\left(\frac{X - \mu}{\sigma}\right)^4\right] = \frac{\mathbb{E}[(X - \mu)^4]}{\sigma^4}.$$

Note that, γ_3 and γ_4 are invariant to changes in location and scale of the distribution.

The skewness is a measure of the asymmetry of the distribution. For a distribution having a symmetric density function about the mean, we have $\gamma_3 = 0$. Otherwise, it is either positive or negative depending on the distribution being *skewed to the right* or *skewed to the left* respectively.

The kurtosis is a measure of the tails of the distribution. As a benchmark, a normal probability distribution (covered in detail in Section 3.6) has $\gamma_4 = 3$. Then, a probability distribution with a higher value of γ_4 can be interpreted as having ‘heavier tails’ (than a normal distribution) while a probability distribution with a lower value is said to have ‘lighter tails’ (than a normal distribution). This benchmark even yields a term called *excess kurtosis* defined as $\gamma_4 - 3$. Hence a positive excess kurtosis implies ‘heavy tails’ and a negative value implies ‘light tails’.

3.3 Functions Describing Distributions

As alluded to in Section 3.2 above, a probability distribution can be described by a probability mass function (PMF) in the discrete case or a probability density function (PDF) in the continuous case. However, there are other popular descriptors of probability distributions, such as the *cumulative distribution function* (CDF), the *complementary cumulative distribution function* (CCDF) and *inverse cumulative distribution function*. Then there are also transform based descriptors including the *moment generating function* (MGF), *probability generating function* (PGF), as well as related functions such as the *characteristic function* (CF), or alternative names, including the *Laplace transform*, *Fourier transform* or *z transform*. Then, for non-negative random variables there is also the *hazard function* which we explore along with the Weibull distribution in Section 3.6. The main point to take away is that a probability distribution can be described in many alternative ways. We now explore a few of these descriptors.

Cumulative Probabilities

Consider first the CDF of a random variable X , defined as,

$$F(x) := \mathbb{P}(X \leq x),$$

where X can be discrete, continuous or a more general random variable. The CDF is a very popular descriptor because unlike the PMF or PDF, it is not restricted to just the discrete or just the continuous case. A closely related function is the CCDF, $\bar{F}(x) := 1 - F(x) = \mathbb{P}(X > x)$.

From the definition of the CDF, $F(\cdot)$,

$$\lim_{x \rightarrow -\infty} F(x) = 0 \quad \text{and} \quad \lim_{x \rightarrow \infty} F(x) = 1.$$

In addition, $F(\cdot)$ is a non-decreasing function. In fact, any function with these properties constitutes a valid CDF and hence a probability distribution of a random variable.

In the case of a continuous random variable, the PDF, $f(\cdot)$ and the CDF, $F(\cdot)$ are related via,

$$f(x) = \frac{d}{dx} F(x) \quad \text{and} \quad F(x) = \int_{-\infty}^x f(t) dt.$$

Also, as a consequence of the CDF properties,

$$f(x) \geq 0, \quad \text{and} \quad \int_{-\infty}^{\infty} f(x) dx = 1. \quad (3.4)$$

Analogously, while less appealing than the continuous counter-part, in the case of discrete random variable, the PMF $p(\cdot)$ is related to the CDF via,

$$p(x) = F(x) - \lim_{t \rightarrow x^-} F(t) \quad \text{and} \quad F(x) = \sum_{k \leq x} p(k). \quad (3.5)$$

Note that here we consider $p(x)$ to be 0 for x not in the support of the random variable. The important point in presenting (3.4) and (3.5) is to show that $F(\cdot)$ is a valid description of the probability distribution.

In Listing 3.5 below, we look at an elementary example, where we consider the PDF $f_2(\cdot)$ of Section 3.1 and integrate it via a crude *Riemann sum* to obtain the CDF:

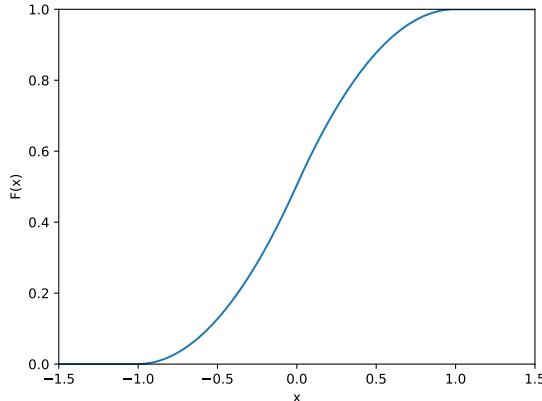
$$F(x) = \mathbb{P}(X \leq x) = \int_{-\infty}^x f_1(u) du \approx \sum_{u=-\infty}^x f(u) \Delta x. \quad (3.6)$$

Listing 3.5: CDF from the Riemann sum of a PDF

```

1  using PyPlot
2
3  f2(x) = (x<0 ? x+1 : 1-x)*(abs(x)<1 ? 1 : 0)
4  a, b = -1.5, 1.5
5  delta = 0.01
6
7  F(x) = sum([f2(u)*delta for u in a:delta:x])
8
9  xGrid = a:delta:b
10 y = [F(u) for u in xGrid]
11 plot(xGrid,y)
12 xlim(a,b)
13 ylim(0,1)
14 xlabel("x")
15 ylabel("F(x)")
```

- In line 3 we define the function `f2()`. Note that the first set of brackets in this equation uses a short circuit style `if` statement to correctly determine which part of the piecewise function should be used. The second set of brackets in the equation are used to ensure that the PDF is only evaluated over the region $[-1, 1]$ (i.e. the second set of brackets acts like the *indicator function*, and evaluates to 0 everywhere else).
- In line 4 and 5 we set the limits of our integral, and the stepwise `delta` used.

Figure 3.4: The CDF associated with the PDF $f_1(x)$.

- In line 7 we create a function that approximates the value of the CDF through a crude Riemann sum by evaluating the PDF at each point u , and then multiplying this by δ , and repeating this process each progressively larger interval up to the specified value x . The total area is then approximated via the `sum` function. See equation (3.6).
- In line 9 we specify the grid of values over which we will plot our approximated CDF.
- Line 10 uses the function `F()` to create the array `y`, which contains the actual approximation of the CDF over the grid of value specified.
- Lines 11-15 plot Figure 3.4.

Inverse and Quantiles

Where the CDF answers the question “what is the probability of being less than or equal to x ”, a dual question often asked is “what value of x corresponds to a probability of the random variable being less than or equal to u ”. Mathematically, we are looking for the *inverse function* of $F(x)$. In cases where the CDF is continuous and strictly increasing over all values, the inverse, $F^{-1}(\cdot)$ is well defined, and can be found via the equation,

$$F(F^{-1}(u)) = u, \quad \text{for } u \in [0, 1]. \quad (3.7)$$

For example, take the *logistic function* as the CDF, which is as a type of *sigmoid function*,

$$F(x) = \frac{1}{1 + e^{-x}}.$$

Solving for $F^{-1}(u)$ in (3.7) yields,

$$F^{-1}(u) = \log \frac{u}{1-u}.$$

This is the *inverse CDF* for the distribution. Schematically, given a specified probability u , it allows us to find x values such that,

$$\mathbb{P}(X \leq x) = u. \quad (3.8)$$

The value x satisfying (3.8) is also called the u 'th *quantile* of the distribution. If u is given as a percent, then it is called a *percentile*. The *median* is another related term, and is also known as the 0.5'th quantile. Other related terms are the *quartiles*, with the *first quartile* at $u = 0.25$, the *third quartile* at $u = 0.75$ and the *inter-quartile range*, which is defined as $F^{-1}(0.75) - F^{-1}(0.25)$. These same terms used again in respect to summarizing datasets in Section 4.2.

In more general cases, where the CDF is not necessarily strictly increasing and continuous, we may still define the inverse CDF via,

$$F^{-1}(u) := \inf\{x : F(x) \geq u\}.$$

As an example of such a case, consider an arbitrary customer arriving to a stochastic queue where the server is utilized 80% of the time, and an average service takes 1 minute. How long does such a customer wait in the queue until service starts? Some customers won't wait at all (20% of the customers), whereas others will need to wait until those that arrived before them are serviced. Results from the field of *queueing theory* (some of which are partially touched in Chapter 9) give rise to the following distribution function for the waiting time:

$$F(x) = 1 - 0.8e^{-(1-0.8)x} \quad \text{for } x \geq 0.$$

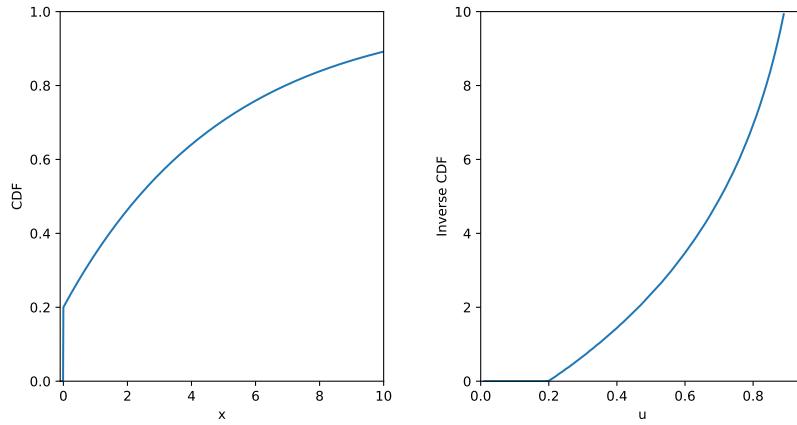
Notice that at $x = 0$, $F(0) = 0.2$, indicating the fact that there is a 0.2 chance for zero wait. Such a distribution is an example of a *mixed discrete and continuous distribution*. Notice that this distribution function only holds for a specific case of assumptions known as the stationary stable M/M/1 queue, explored further in Section 10.3.

We now plot both $F(x)$ and $F^{-1}(u)$ in Listing 3.6 below, where we construct $F^{-1}(\cdot)$ programmatically.

Listing 3.6: The inverse CDF

```

1  using PyPlot
2
3  xGrid = 0:0.01:10
4  uGrid = 0:0.01:1
5  busy = 0.8
6
7  F(t) = t<=0 ? 0 : 1 - busy*exp(-(1-busy)*t)
8
9  infimum(B) = isempty(B) ? Inf : minimum(B)
10 invF(u) = infimum(filter((x) -> (F(x) >= u), xGrid))
11
12 figure(figsize=(10,5))
13 subplots_adjust(wspace=0.3)
14
15 subplot(121)
16 plot(xGrid,F.(xGrid))
17 xlim(-0.1,10)
18 ylim(0,1)
19 xlabel("x")
20 ylabel("CDF")
21
22 subplot(122)
23 plot(uGrid,invF.(uGrid))
24 xlim(0,0.95)
25 ylim(0,maximum(xGrid))
```

Figure 3.5: The CDF $F(x)$, and its inverse $F^{-1}(u)$.

```

26 xlabel("u")
27 ylabel("Inverse CDF")

```

- In Line 3 defines the grid over which we will evaluate the CDF.
- Line 4 defines the grid over which we will evaluate the inverse CDF. Note that the domain here is $[0, 1]$. Values selected from this domain are also known as the quantiles of our distribution.
- In line 5 we define the proportion of times that there is someone already being served.
- In line 7 we define the function `F()` as given above. Note that for values less than zero, the CDF evaluates to 0.
- In line 9 we define the function `infimum()`, which implements similar logic to the mathematical operation `inf{}`. It takes an input and checks if it is empty via the `isempty()` function, and if it is, returns `Inf`, else returns the minimum value of the input. This agrees with the typical mathematical notation where the infimum of the empty set is ∞ .
- In line 10 we define the function `invF()`. It first creates an array (representing a set) $\{x : F(x) \geq u\}$ directly via the Julia `filter()` function. Note that as a first argument, we use an anonymous Julia function, `(x) -> (F(x) >= u)`. We then use this function as a filter over `xGrid`. Finally we apply the infimum over this mathematical set (represented by a vector of coordinates on the x axis).
- Lines 12-27 are used to plot both the original CDF via the `F()` function, and the inverse CDF via the `invF()` functions respectively. Observe Figure 3.5. The CDF, $F(x)$ exhibits a jump at 0 indicating the “probability mass”. The inverse CDF then returns 0 for all values of $u \in [0, 0.2]$.

Integral Transforms

In general terms, an *integral transform* of a probability distribution is a representation of the distribution on a different domain. Here we focus on the moment generating function (MGF).

Other examples include the characteristic function (CF), probability generating function (PGF) and similar transforms.

For a random variable X and a real or complex fixed value s , consider the expectation, $\mathbb{E}[e^{sX}]$. When viewed as a function of s , this is the moment generating function. We present this here for such a continuous random variable with PDF $f(\cdot)$:

$$M(s) = \mathbb{E}[e^{sX}] = \int_{-\infty}^{\infty} f(x) e^{sx} dx. \quad (3.9)$$

This is also known as the bi-lateral *Laplace transform* of the PDF (with argument $-s$). Many of useful Laplace transform properties carry over from the theory of Laplace transforms to the MGF. A full exposition of such properties are beyond the scope of this book, however we illustrate a few via an example.

Consider two distributions with densities,

$$\begin{aligned} f_1(x) &= 2x && \text{for } x \in [0, 1]. \\ f_2(x) &= 2 - 2x && \text{for } x \in [0, 1], \end{aligned}$$

where the respective random variables are denoted X_1 and X_2 . Computing the MGF of these distributions we obtain,

$$\begin{aligned} M_1(s) &= \int_0^1 2x e^{sx} dx = 2 \frac{1 + e^s(s-1)}{s^2}, \\ M_2(s) &= \int_0^1 (2 - 2x) e^{sx} dx = 2 \frac{e^s - 1 - s}{s^2}. \end{aligned}$$

Define now a random variable, $Z = X_1 + X_2$ where X_1 and X_2 are assumed independent. In this case, it is known that the MGF of Z is the product of the MGFs of X_1 and X_2 . That is,

$$M_Z(s) = M_1(s)M_2(s) = 4 \frac{(1 + e^s(s-1))(e^s - 1 - s)}{s^4}.$$

The new MGF, $M_Z(\cdot)$ fully specifies the distribution of Z . It also yields a rather straightforward computation of moments (hence the name MGF). A key property of any MGF $M(s)$ of a random variable X , is that,

$$\frac{d^n}{ds^n} M(s) \Big|_{s=0} = \mathbb{E}[X^n]. \quad (3.10)$$

This can be easily verified from (3.9). Hence to calculate the n 'th moment, it is simply required to evaluate the derivative of the MGF at $s = 0$.

In Listing 3.7 below, we estimate both the PDF and MGF of Z and compare the estimated MGF to $M_Z(s)$ above.

Listing 3.7: A sum of two triangular random variables

```

1  using Distributions, PyPlot
2
3  dist1 = TriangularDist(0, 1, 1)

```

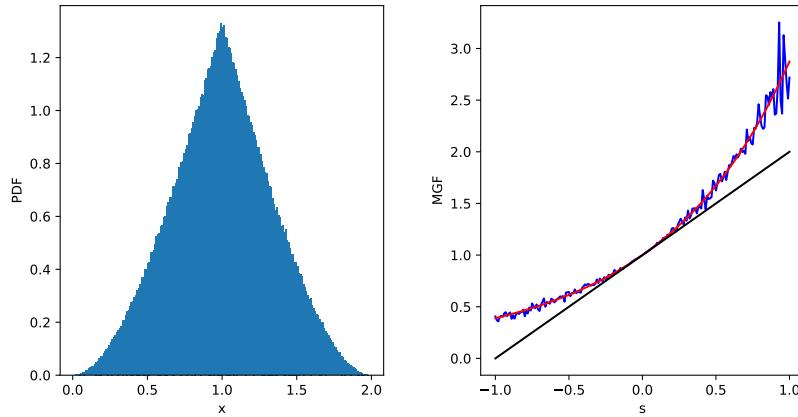


Figure 3.6: Left: The estimate of the PDF of Z via a histogram.
Right: The theoretical MGF in red vs. a Monte Carlo estimate in blue.
The slope of the black line is the mean.

```

4 dist2 = TriangularDist(0,1,0)
5 N=10^6
6
7 data1, data2 = rand(dist1,N),rand(dist2,N)
8 dataSum = data1 + data2
9
10 figure(figsize=(10,5))
11 subplots_adjust(wspace=0.3)
12
13 subplot(121)
14 p2 = plt[:hist](dataSum,200,normed="true");
15 xlabel("x")
16 ylabel("PDF")
17
18 sGrid = -1:0.01:1
19 mgf(s) = 4(1+(s-1)*MathConstants.e^s)*(MathConstants.e^s-1-s)/s^4
20
21 function mgfEst(s)
22     N = 20
23     data1, data2 = rand(dist1,N),rand(dist2,N)
24     dataSum = data1 + data2
25     sum([MathConstants.e^(s*x) for x in dataSum])/length(dataSum)
26 end
27
28 subplot(122)
29 plot(sGrid,mgfEst.(sGrid),"b")
30 plot(sGrid,mgf.(sGrid),"r")
31 plot([minimum(sGrid),maximum(sGrid)],[minimum(sGrid),maximum(sGrid)].+1,"k")
32 xlabel("s")
33 ylabel("MGF");

```

- In lines 3 and 4 we create two separate triangular distribution type objects $dist1$ and $dist2$, for our two densities $f_1(x)$ and $f_2(x)$ respectively. Note that the third argument of the `TriangularDist()` function is location of the “peak” of the triangle. More on using such distribution objects is in Section 3.4 below.

- In line 7 we generate random observations from `dist1` and `dist2`, and store these observations separately in the two arrays `data1` and `data2` respectively.
- In line 8 we generate observations for Z by performing element-wise summation of the values in our arrays `data1` and `data2`.
- Lines 13-16 plot a histogram of Z with 200 bins. Note that it is possible to obtain this curve analytically by carrying out a *convolution* of the triangular PDFS (we don't do this here).
- Line 19 defines the function `mgf()`. This is exactly $M_Z(s)$ as developed above.
- In lines 21-26 we define the function `mgfEst()`. It is a crude way to estimate the MGF at the point s . The key line is line 25 where we carry out an expectation estimate in a straight forward manner. Note that we purposefully choose a small number of observations ($N=20$). This is to see the variability of the estimate for varying values of s . In practice, if estimating an MGF, it is generally better to use one set of common random numbers for all values of s .
- Lines 28-32 plot the theoretical MGF (red) vs. the estimated MGF (blue) on the interval $[-1, 1]$. Notice that an MGF always equals 1 at $s = 0$. In practice, looking at the actual curve of the MGF (as a function of a real variable s) is not common. Unlike the PDF that presents us with visual intuitive information about the distribution, the MGF's graph does not immediately do that. It is rather the infinite sequence of derivatives at $s = 0$ that capture the information of the distribution via moments. These are clearly not fully visible from the figure. An exception is the diagonal black line ($y = x + 1$) with slope 1 that we overlay on the figure. The slope of 1 agrees with the fact that the mean of the distribution is 1. You can see that this line is the tangent line to the MGF at $s = 0$ (see equation (3.10) with $n = 1$). In the code we plot it using code line 31; here "k" indicates to plot in black.

3.4 The Distributions Package

As touched on previously in Listing 3.7 and Listing 3.4, Julia has a well developed package for distributions. The `Distributions` package allows us to create distribution type objects based on what family they belong to (more on families of distributions in the sequel). These distribution objects can then be used as arguments for other functions, for example `mean()` and `var()`. Of key importance is the ability to randomly sample from a distribution using `rand()`. We can also use distributions with the functions `pdf()`, `cdf()` and `quantile()` to name a few.

In addition, the builtin `Statistics` package as well as the `StatsBase` packages contain many functions which aid distribution type objects. It contains functions useful when it comes to summarizing data. This aspect is covered further in Section 4.2. We now explore some of the applications of the functions from these two packages below.

Weighted Vectors

In the case of discrete distributions (of finite support), the `StatsBase` package provides the “weight vector” object via `Weights()`, which allows for an array of values (i.e. outcomes) to

be given probabilistic weights. This is also known as a *probability vector*. We can then combine these with functions such as `mean()` and `variance()`. In order to generate observations we use the `sample()` function (from `StatsBase`) on a vector given its weights, instead of the `rand()` function.

Listing 3.8 below provides a brief example of the use of weight vectors.

Listing 3.8: Sampling from a weight vector

```

1  using StatsBase
2
3  grade = ["A", "B", "C", "D", "E"]
4  weightVect = Weights([0.1, 0.2, 0.1, 0.2, 0.4])
5
6  N = 10^6
7  data = sample(grade, weightVect, N)
8  [count(i->(i==g), data) for g in grade]/N

```

```

5-element Array{Float64,1}:
 0.099673
 0.200357
 0.099714
 0.200204
 0.400052

```

- Line 1 loads the `StatsBase` package, which contains the functions `Weights()` and `sample()` which are required for this example. Note the fact that `Weights()` is capitalized, signifying the fact that the function creates a new object. This type of function is known as a *Constructor*.
- In line 3 we define an array of strings “A” to “E”.
- In line 4 we use the `Weights()` function to create a weight vector based on the weights given. Note that five weights are specified, one for each of the grades, and that the sum of all weights is 1. If the weights do not sum to 1, then the weightings are automatically normalized proportionally.
- Line 7 uses the function `sample()` to sample `N` observations from our array `grade`, according to the weights given by the weight vector `weightVect`.
- Line 8 uses the `count()` function to count how many times each entry `g` in `grade` occurs in `data`, and then returns the proportion of times total each grade occurs. It can be observed that the grades have been sampled according to the probabilities specified in the array `weightVect`.

Using Distribution Type Objects

We now introduce some important functionality of the `Distributions` package and distribution type objects through an example. Consider a distribution from the “Triangular” family, with

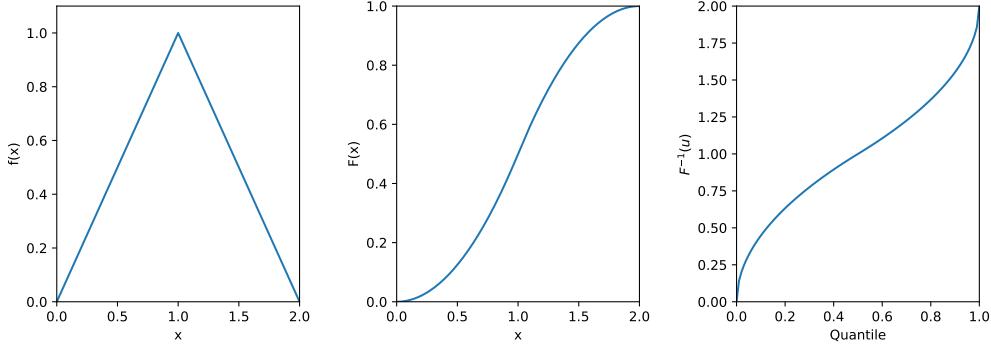


Figure 3.7: The PDF, CDF and inverse CDF a triangular distribution.

the following density,

$$f(x) = \begin{cases} x & \text{if } x \in [0, 1], \\ 2 - x & \text{if } x \in (1, 2]. \end{cases}$$

In Listing 3.9 below, rather than creating the density manually as in the previous sections, we use the `TriangularDist()` function to create a distribution type object, and then use this to create plots of the PDF, CDF and inverse CDF.

Listing 3.9: Using the `pdf()`, `cdf()`, and `quantile()` functions with Distributions

```

1  using Distributions, PyPlot
2
3  tDist = TriangularDist(0, 2, 1)
4  xGrid = 0:0.01:2
5  xGrid2 = 0:0.01:1
6
7  figure(figsize=(12.4, 4))
8  subplots_adjust(wspace=0.4)
9
10 subplot(131)
11 plot(xGrid,pdf(tDist,xGrid))
12 xlim(0,2)
13 ylim(0,1.1)
14 xlabel("x")
15 ylabel("f(x)")
16
17 subplot(132)
18 plot(xGrid,cdf(tDist,xGrid))
19 xlim(0,2)
20 ylim(0,1)
21 xlabel("x")
22 ylabel("F(x)")
23
24 subplot(133)
25 plot(xGrid2,quantile(tDist,xGrid2))
26 xlim(0,1)
27 ylim(0,2)
28 xlabel("Quantile")
29 ylabel(L"$F^{-1}(u)$");

```

- In line 3 we use the `TriangularDist()` function to create a distribution type object. The first two arguments are the start and end points of the distribution, and the third argument is the location of the “peak” of the triangular distribution.
- In line 4 we create the grid of values over which we plot the PDF and CDF of our distribution, while in line 5 we create the grid over which we will plot the inverse CDF. Note that the latter is over the domain $[0, 1]$ and is the range of quantiles of our distribution.
- Lines 7-29 are used to create Figure 3.7. There are however a few key lines which we elaborate on now.
 - In line 11 we use the `pdf()` function on the distribution `tDist` to evaluate all values of the PDF for each point in `xGrid`.
 - In line 18 we use the `cdf()` function on the distribution `tDist` to evaluate all values of the CDF for each point in `xGrid`.
 - In line 25 we use the `quantile()` function on the distribution `tDist` to evaluate all values of the PDF for each point in `xGrid2`.

In addition to evaluating functions associated with the distribution, we can also query a distribution object for a variety of properties and parameters. Given a distribution object, you may apply `params()` on it to retrieve the distributional parameters; you may query for the `mean()`, `median()`, `var()` (variance), `std`, (standard deviation), `skewness()` and `kurtosis()`. You can also query for the minimal and maximal value in the support of the distribution via `minimum()` and `maximum()` respectively. You may also apply `mode()` or `modes()` to either get a single mode (value of x where the PMF or PDF is maximized) or an array of modes where applicable.

The short listing below illustrates this for our `TriangularDist`:

Listing 3.10: Descriptors of Distributions objects

```

1  using Distributions
2  tDist = TriangularDist(0,2,1)
3
4  println("Parameters: \t\t\t",params(tDist))
5  println("Central descriptors: \t\t\t",mean(tDist),"\t",median(tDist))
6  println("Dispersion descriptors: \t\t\t", var(tDist),"\t",std(tDist))
7  println("Higher moment shape descriptors: ", skewness(tDist),"\t",kurtosis(tDist))
8  println("Range: \t\t\t\t\t", minimum(tDist),"\t",maximum(tDist))
9  println("Mode: \t\t\t\t\t", mode(tDist), "\tModes: ",modes(tDist))

```

Parameters:	(0.0, 2.0, 1.0)	
Central descriptors:	1.0 1.0	
Dispersion descriptors:	0.1666666666666666	0.408248290463863
Higher moment shape descriptors:	0.0 -0.6	
Range:	0.0 2.0	
Mode:	1.0 Modes: [1.0]	

In Listing 3.11 below we look at another example, where we generate random observations from a distribution type object via the `rand()` function, and compare the sample mean against the specified mean. Note that two different types of distributions are created here, a continuous distribution and discrete distribution. These are discussed further in Sections 3.5 and 3.6 respectively.

Listing 3.11: Using rand() with Distributions

```

1  using Distributions, StatsBase
2
3  dist1 = TriangularDist(0,10,5)
4  dist2 = DiscreteUniform(1,5)
5  theorMean1, theorMean2 = mean(dist1), mean(dist2)
6
7  N=10^6
8  data1 = rand(dist1,N)
9  data2 = rand(dist2,N)
10 estMean1, estMean2 = mean(data1), mean(data2)
11
12 println("Symmetric Triangular Distribution on [0,10] has mean $theorMean1
13           (estimated: $estMean1)")
14 println("Discrete Uniform Distribution on {1,2,3,4,5} has mean $theorMean2
15           (estimated: $estMean2)")

```

Symmetric Triangular Distribution on [0,10] has mean 5.0 (estimated: 4.998652531225146)
 Discrete Uniform Distribution on {1,2,3,4,5} has mean 3.0 (estimated: 2.998199)

- In line 3 we use the `TriangularDist()` function to create a symmetrical triangular distribution about 5, and store this as `dist1`.
- In line 4 we use the `DiscreteUniform()` function to create a discrete uniform distribution, and store this as `dist1`. Note that observations from this distribution can take on values from $\{1, 2, 3, 4, 5\}$
- In line 5 we evaluate the mean of the two distribution objects created above by applying the function `mean()` to both of them.
- In lines 7-10 we estimate the means of the two distributions by randomly sampling from our distributions `dist1` and `dist2`.
- In line 7 we set how many observations we will make from each distribution.
- In lines 8 and 9, we randomly sample from these distributions, and store the results in the arrays `data1` and `data2` respectively.
- In line 10 we calculate the means of the data from our two arrays `data1` and `data2`.
- Lines 12-15 print the results. It can be seen that the estimated means are a good approximation of the actual means.

The Inverse Probability Transform

We now look at the concept known as *inverse transform sampling*. Let X be a random variable distributed with CDF $F(\cdot)$ and inverse CDF $F^{-1}(\cdot)$. Now take U to be a uniform random variable over $[0, 1]$, and let $Y = F^{-1}(U)$. It holds that Y is distributed like X . This useful property is called the *inverse probability transform*. It constitutes a generic method for generating random variables from an underlying distribution.

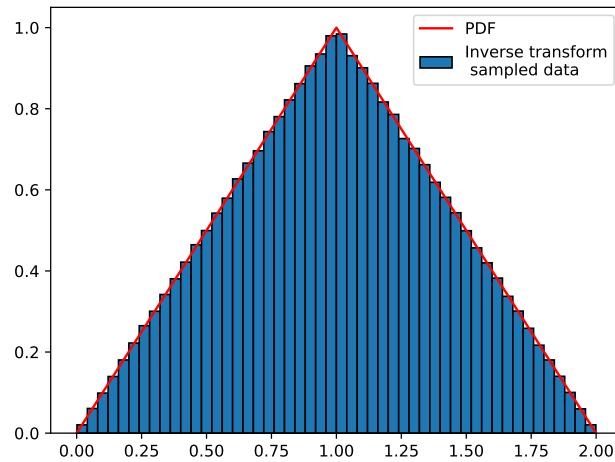


Figure 3.8: A histogram generated using the inverse probability transform compared to the PDF of a triangular distribution.

To see this, consider a uniform random variable U and apply to it the inverse probability transform $F^{-1}(\cdot)$. In such a case, consider the CDF of $Y = F^{-1}(U)$ and see that it is $F(\cdot)$:

$$F_Y(y) = \mathbb{P}(y \leq Y) = \mathbb{P}(y \leq F^{-1}(U)) = \mathbb{P}(F(y) \leq U) = F_U(F(y)) = F(y).$$

The last step follows because the CDF of uniform $(0, 1)$ random variable is,

$$F_U(y) = \begin{cases} 0, & y < 0, \\ y, & y \in [0, 1], \\ 1, & 1 < y. \end{cases}$$

Keep in mind, that when using the Distributions package, we would typically generate random variables using the `rand()` function on a distribution type object, as performed in Listing 3.11 above. However, in Listing 3.12 below, we illustrate how to use the inverse probability transform. Observe that we can implement $F^{-1}(\cdot)$ via the `quantile()` function.

Listing 3.12: Inverse transform sampling

```

1  using Distributions, PyPlot
2
3  triangDist = TriangularDist(0,2,1)
4  xGrid = 0:0.1:2
5  N = 10^6
6  inverseSampledData = quantile.(triangDist,rand(N))
7
8  plt[:hist](inverseSampledData,50, normed = true, ec="k",
9             label="Inverse transform\n sampled data")
10 plot(xGrid,pdf(triangDist,xGrid),"r",label="PDF")
11 legend(loc="upper right")

```

- In line 3 we create our triangular distribution `triangDist`.

- In lines 4 and 5 we define the support over which we will plot our data, as well as how many data-points we will simulate.
- In line 6 we generate N random observations from a continuous uniform distribution over the domain $[0, 1)$ via the `rand()` function. We then use the `quantile()` function, along with the `dot` operator (`.`) to calculate each corresponding quantile of `triangDist`.
- Lines 8 and 9 plot a histogram of this `inverseSampledData`, using 50 bins. For large N, the histogram generated is a close approximation of the PDF of the underlying distribution.
- Line 10 then plots the analytic PDF of the underlying distribution.

3.5 Families of Discrete Distributions

A *family of probability distributions* is a collection of probability distributions having some functional form that is parameterized by a well defined set of parameters. In the discrete case, the PMF, $p(x; \theta) = \mathbb{P}(X = x)$, is parameterized by the *parameter* $\theta \in \Theta$ where Θ is called the *parameter space*. The (scalar or vector) parameter θ then affects the actual form of the PMF including possibility the support of the random variable. Hence, technically a family of distributions is the collection of PMFs $p(\cdot; \theta)$ for all $\theta \in \Theta$.

In this section we present some of the most common families of *discrete distributions*. We consider the following: *discrete uniform distribution*, *binomial distribution*, *geometric distribution*, *negative binomial distribution*, *hypergeometric distribution* and *poisson distribution*. Each of these is represented in the Julia Distributions package. The approach that we take in the code examples below is to generate random variables from each such distribution using first principles as opposed to applying `rand()` on a distribution object, as demonstrated in Listing 3.11 above. Understanding how to generate a random variable from a given distribution using first principles helps strengthen understanding of the associated probability models and processes.

In Listing 3.13 below we illustrate how to create a distribution object for each of the discrete distributions that we investigate in the sequel. As output we print the parameters and the support of each distribution.

Listing 3.13: Families of discrete distributions

```

1  using Distributions
2  dists = [
3      DiscreteUniform(10,20),
4      Binomial(10,0.5),
5      Geometric(0.5),
6      NegativeBinomial(10,0.5),
7      Hypergeometric(30, 40, 10),
8      Poisson(5.5)];
9
10 println("Distribution \t\t\t Parameters \t Support")
11 reshape([dists ; params.(dists) ;
12           ((d)->(minimum(d),maximum(d))).(dists) ],
13           length(dists),3)

```

- Lines 2-8 are used to define an array of distribution objects. For each such distribution feel free to use ? «Name» where «Name» may be DiscreteUniform, Binomial, etc... The help provided by the distributions package may yield more information.
- Lines 11-13 result in output that is a 6×3 array of type Any. The first column is the actual distributions object, the second column represents the parameters and the third column represents the support. More details on the parameters and the support for each distribution are presented in the sequel. Note the use of an anonymous function $(d) \rightarrow (\text{minimum}(d), \text{maximum}(d))$ applied via ‘.’ to each element of `dists`. This function creates a tuple as a return argument. The use of `reshape()` transforms the array of arrays into a matrix of the desired dimensions.

Discrete Uniform Distribution

The *discrete uniform distribution* is simply a probability distribution that places equal probabilities for all equal outcomes. One example is given by the probability of the outcomes of a dice toss. The probability of each possible outcome for a fair, six-sided die is given by,

$$\mathbb{P}(X = x) = \frac{1}{6} \quad \text{for } x = 1, \dots, 6.$$

Listing 3.14 below simulates N dice tosses, and then calculates and plots the proportion of times each possible outcome occurs, along with the analytic solution of the PDF. For large values of N , the proportion of counts for each outcome converges to $1/6$.

Listing 3.14: Discrete uniform dice toss

```

1  using StatsBase, PyPlot
2
3  faces, N = 1:6, 10^6
4  mcEstimate = counts(rand(faces,N), faces)/N
5  stem(faces,mcEstimate,basefmt="none",label="MC estimate")
6  plot([i for i in faces],[1/6 for _ in faces],"rx",label="PMF")
7  xlabel("Face number")
8  ylabel("Probability")
9  ylim(0,0.2);
10 legend(loc="upper right");

```

- In line 3 we define all possible outcomes of our six-sided die, along with how many die tosses we will simulate.
- Line 4 uniformly and randomly generates N observations from our die, and then uses the `counts()` function to calculate proportion of times each outcome occurs. Note that applying `rand(DiscreteUniform(1, 6), N)` would yield a statistically identical result to `rand(faces, N)`.
- Line 5 uses the `stem` function to create a stem plot of the proportion of times each outcome occurs.
- Line 6 plots the analytic PMF of our six-sided die. Notice that For large N , we observe the simulated estimate is a good approximation of the PMF.

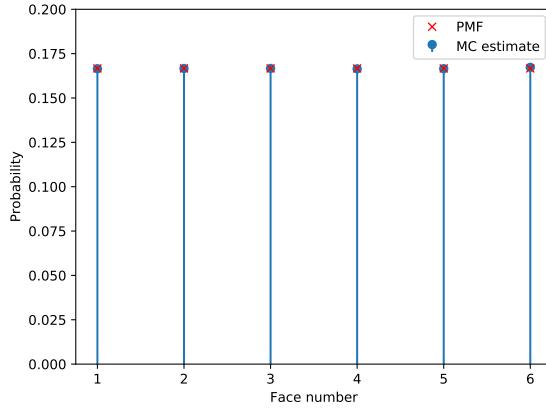


Figure 3.9: A discrete uniform PMF.

Binomial Distribution

The *binomial distribution* is a discrete distribution which arises where multiple identical and independent yes/no, true/false, success/failure trials (also known as *Bernoulli trials*) are performed. For each trial, there can only be two outcomes, and the probability weightings of each unique trial must be the same.

As an example, consider a two-sided coin, which is flipped n times in a row. If the probability of obtaining a head in a single flip is p , then the probability of obtaining x heads total is given by the PMF,

$$\mathbb{P}(X = x) = \binom{n}{x} p^x (1-p)^{n-x} \quad \text{for } x = 0, 1, \dots, n.$$

Listing 3.15 below simulates 10 tosses of a coin, N times total, with success probability p , and calculates the proportion of times each possible outcome occurs.

Listing 3.15: Coin flipping and the binomial distribution

```

1  using StatsBase, Distributions, PyPlot
2
3  function binomialRV(n,p)
4      return sum(rand(n) .< p)
5  end
6
7  p, n, N = 0.5, 10, 10^6
8
9  bDist = Binomial(n,p)
10 xGrid = 0:n
11 bPmf = [pdf(bDist,i) for i in xGrid]
12 data = [binomialRV(n,p) for _ in 1:N]
13 pmfEst = counts(data,0:n)/N
14
15 stem(xGrid,pmfEst,label="MC estimate",basefmt="none")
16 plot(xGrid,bPmf,"rx",ms=8,label="PMF")
17 ylim(0,0.3)
18 xlabel("x")
19 ylabel("Probability")
20 legend(loc="upper right");

```

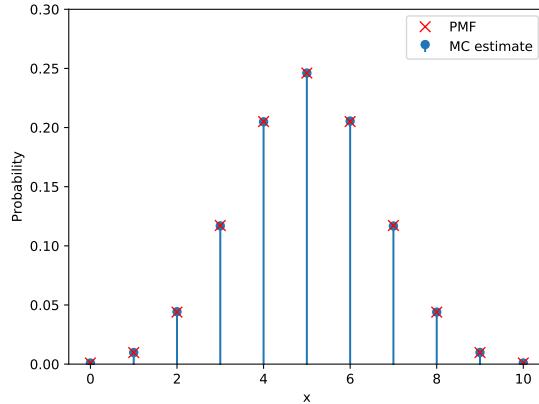


Figure 3.10: Binomial PMF for number of heads in 10 flips each with $p = 0.5$.

- In lines 3-5, our function `binomialRV()` generates a binomial random variable from first principles. In line 4 we create an array of uniform $[0, 1]$ values of length n with `rand(n)`. We then use `.<` to compare each value (element-wise) to p . The result is a vector of booleans, with each one set to `true` with probability p . Summing up this vector creates the binomial random variable.
- In line 11 we create a vector incorporating the values of the binomial PMF. Note that in the Julia distributions package, PMFs are created via `pdf()`.
- Line 12 is where we generate N values of random values.
- In line 13 we use `counts()` from the `StatsBase` package to count how many elements were in each value of the support $(0:n)$. We then normalize via division by N .
- The remaining parts of the code create the plot, showing that the estimated values are at the expected theoretical values predicted by the PMF.

Note that the Binomial distribution describes part of the fishing example in Section 2.1, where we sample with replacement. This is because the probability of success (i.e. fishing a gold fish) remains unchanged regardless of how many times we have sampled from the pond.

Geometric Distribution

Another distribution associated with Bernoulli trials is the *geometric distribution*. In this case, consider an infinite sequence of independent trials, each with success probability p , and let X be the first trial that is successful. Using first principles it is easy to see that the PMF is,

$$\mathbb{P}(X = x) = p(1 - p)^{x-1} \quad \text{for } x = 1, 2, \dots \quad (3.11)$$

An alternative version of the geometric distribution is the distribution of the random variable, \tilde{X} counting the number of failures until success. Observe that for every sequence of trials, $\tilde{X} = X - 1$. From this it is easy to relate the PMFs of the random variables and see that,

$$\mathbb{P}(\tilde{X} = x) = p(1 - p)^x \quad \text{for } x = 0, 1, 2, \dots$$

In the Julia Distributions package, Geometric stands for the distribution of \tilde{X} , not X .

We now look at an example involving the popular casino game of roulette. Roulette is a game of chance, where a ball is spun on the inside edge of a horizontal wheel. As the ball loses momentum, it eventually falls vertically down, and lands on one of 37 spaces, numbered 0 to 36. 18 spaces are black, 18 red, and a single space (zero) is green. Each spin of the wheel is independent, and each of the possible 37 outcomes is equally likely. Now let us assume that a gambler goes to the casino and plays a series of roulette spins. There are various ways to bet on the outcome of roulette, but in this case he always bets on black (if the ball lands on black he wins, otherwise he loses). Say that the gambler plays until his first win. In this case, the number of plays is a geometric random variable. Listing 3.16 simulates this scenario.

Listing 3.16: The geometric distribution

```

1  using StatsBase, Distributions, PyPlot
2
3  function rouletteSpins(p)
4      x = 0
5      while true
6          x += 1
7          if rand() < p
8              return x
9          end
10     end
11 end
12
13 p, xGrid, N = 18/37, 1:7, 10^6
14
15 mcEstimate = counts([rouletteSpins(p) for _ in 1:N],xGrid)/N
16
17 gDist = Geometric(p)
18 gPmf = [pdf(gDist,x-1) for x in xGrid]
19
20 stem(xGrid,mcEstimate,label="MC estimate",basefmt="none")
21 plot(xGrid,gPmf,"rx",ms=8,label="PMF")
22 ylim(0,0.5)
23 xlabel("x")
24 ylabel("Probability")
25 legend(loc="upper right");

```

- The function `rouletteSpins()` defined on lines 3-11 is a straight forward way to generate a geometric random variable (with support $1, 2, \dots$ as X above). In the function we loop on lines 6-9 until a value is returned from the function. In each iteration, we increment `x` and check if we have a success (an event happening with probability p) via, `rand() < p`.
- The remainder of the code is similar to the previous example. Now however the second argument to `pdf()` on line 18. Here we use `x-1` because the built in geometric distribution is for the random variable \tilde{X} above (starting at 0), while we are interested in the Geometric random variable starting at 1.

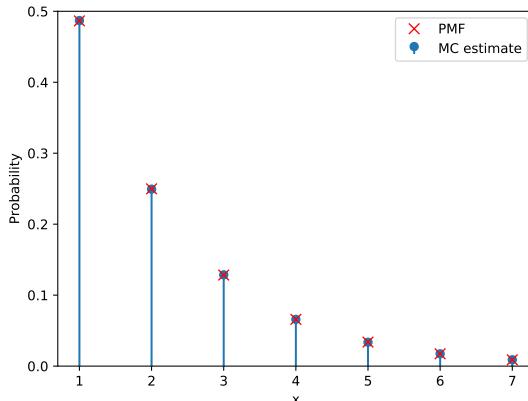


Figure 3.11: A geometric PMF.

Negative Binomial Distribution

Recall the example above of a roulette gambler. Assume now that the gambler plays until he wins for the r 'th time (in the previous example $r = 1$). The *negative binomial distribution* describes this situation. That is, a random variable X follows this distribution, if it describes the number of trials up to the r 'th success. The PMF is given by

$$\mathbb{P}(X = x) = \binom{x-1}{r-1} p^r (1-p)^{x-r} \quad \text{for } x = r, r+1, r+2, \dots$$

Notice that with $r = 1$ the expression reduces to the geometric PMF, (3.11). Similarly to the geometric case, there is an alternative version of the negative binomial distribution. Let \tilde{X} denote the number of failures until the r 'th success. Here, like in the geometric case, when both random variables are coupled on the same sequence of trials, we have, $\tilde{X} = X - r$.

$$\mathbb{P}(\tilde{X} = x) = \binom{x+r-1}{x} p^r (1-p)^x \quad \text{for } x = 0, 1, 2, \dots$$

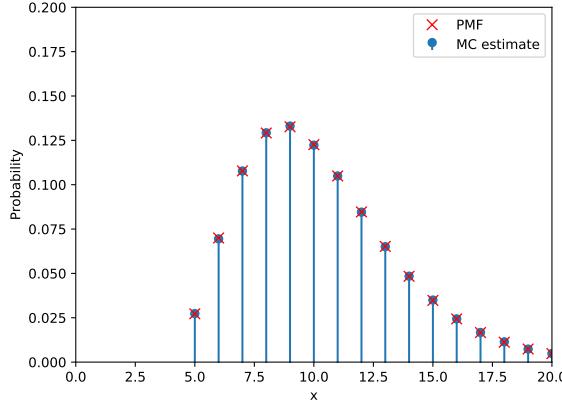
To help reinforce this, in Listing 3.17 below we simulate a gambler who bets consistently on black much like in the previous scenario, and determine the PMF for $k = 3$. That is, we determine the probabilities that x failures will occur before the 3'rd success (or win).

Listing 3.17: The negative binomial distribution

```

1  using StatsBase, Distributions, PyPlot
2
3  function rouletteSpins(r,p)
4      x = 0
5      wins = 0
6      while true
7          x += 1
8          if rand() < p
9              wins += 1
10             if wins == r
11                 return x
12             end

```

Figure 3.12: The PMF of negative binomial with $r = 5$ and $p = 18/37$.

```

13         end
14     end
15 end
16
17 r, p, N = 5, 18/37, 10^6
18 xGrid = r:r+15
19
20 mcEstimate = counts([rouletteSpins(r,p) for _ in 1:N],xGrid)/N
21
22 nbDist = NegativeBinomial(r,p)
23 nbPmf = [pdf(nbDist,x-r) for x in xGrid]
24
25 stem(xGrid,mcEstimate,label="MC estimate",basefmt="none")
26 plot(xGrid,nbPmf,"rx",ms=8,label="PMF")
27 xlim(0,maximum(xGrid))
28 ylim(0,0.2)
29 xlabel("x")
30 ylabel("Probability")
31 legend(loc="upper right");

```

- This code is similar to the previous listing. The modification is the function `rouletteSpins()` which nows accepts both `r` and `p` as arguments. It is a straight forward implementation of the negative binomial story. A value is returned (line 11) only once the number of wins equals `r`.
- In a similar manner to he geometric example notice that on line 23 we use `x-r` for the argument of the `pdf()` function. This is because `NegativeBinomial` in the `Distributions` package stands for a distribution with support, $x = 0, 1, 2, \dots$ and not $x = r, r + 1, r + 2, \dots$ as we desire.

Hypergeometric Distribution

Moving away from Bernoulli trials, we now consider at the *hypergeometric distribution*. To put it in context, consider the fishing problem discussed in Section 2.1, specifically consider the case where

we fish without replacement. In this scenario, each time we sample from the population it decreases and hence the probability of success changes for each subsequent sample. The hypergeometric distribution describes this situation with a PMF given by,

$$p(x) = \frac{\binom{K}{x} \binom{L-K}{n-x}}{\binom{L}{n}} \quad \text{for } x = \max(0, n+K-L), \dots, \min(n, K).$$

Here the parameter L is the population size, the parameter K is the number of successes present in the population (this implies that $L - K$ is the number of failures present in the population). The parameter n is the number of samples taken from the population, and the input argument x is the number of successful samples observed. Hence a hypergeometric random variable X with $\mathbb{P}(X = x) = p(x)$ describes the number of successful samples when *sampling without replacement*. Note that the expression for $p(x)$ can be deduced directly by combinatorial counting arguments.

To understand the support of the distribution consider first the least possible value, $\max(0, n + K - L)$. It is either 0 or $n + K - L$ in case $n > L - K$. The latter case stems from a situation where the number of samples n , is greater than the number of failures present in the population. That is, in such a case the least possible number of successes that can be sampled is,

$$\text{number of samples } (n) - \text{number of failures in the population } (L - K).$$

As for the upper value of the support, it is $\min(n, K)$ because if $K < n$ then it isn't possible that all samples are successes. Note that in general if the sample size, n is not "too big" then the support reduces to $x = 0, \dots, n$.

To help illustrate this distribution, we look at an example where we compare several hypergeometric distributions simultaneously. As before, let us consider a pond which contains a combination of gold and silver fish. In this example, there are $N = 500$ fish total, and we will define the catch of a gold fish a success, and a silver fish a failure. Now say that we sample $n = 30$ of them without replacement. Here we consider several of these examples, where the only difference between each is the number of successes, K , (gold fish) in the population.

Notice that in the Julia Distributions package, `Hypergeometric` is parameterized via the number of successes (first argument) and number of failures (second argument), with the third argument being the sample size . This is slightly different than our parameterization above via N , K and n .

Listing 3.18 below plots the PMF's of 5 different hypergeometric distributions based on the number of successes in the population.

Listing 3.18: Comparison of several hypergeometric distributions

```

1  using Distributions, PyPlot
2
3  L, K, n  = 500, [450, 400, 250, 100, 50], 30
4  hyperDists = [Hypergeometric(k,L-k,n) for k in K]
5  xGrid = 0:1:30
6
7  [bar(xGrid, pdf(hyperDists[i],xGrid), width=0.2, alpha=0.65,

```

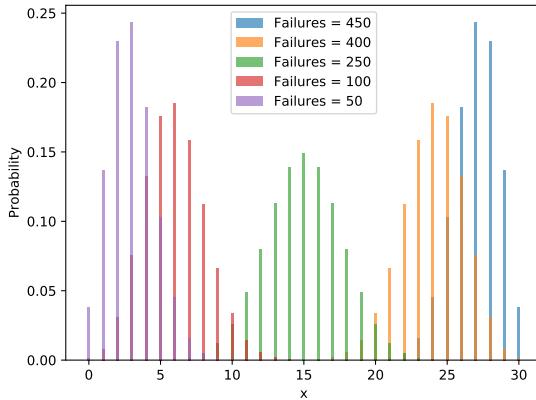


Figure 3.13: A comparison of several hypergeometric distributions for different proportions in a population.

```

8      label="Failures = $(K[i])") for i in 1:length(K)
9      legend()
10     xlabel("x")
11     ylabel("Probability");

```

- In line 3 we define our population size, L, and the array K, which contains the number of successes/failures in our population, for each of our 5 scenarios.
- In line 4 we use the `Hypergeometric()` function to create several hypergeometric distributions. The function takes three arguments, the number of successes in the population n, the number of failures in the population population-k, and the number of times we sample from the population without replacement n. This function is then wrapped in a comprehension in order to create an array of different hypergeometric distributions, `hyperDists`, with the only difference being the number of failures present in the population, with the values taken from K previously defined.
- In lines 7-8, the `bar()` function is used to plot a bar chart of the analytic PMF of each hypergeometric distribution in `hyperDists`.
- It can be observed that as the number of failures present in the population increases, the PMF shifts further towards the right. This represents the fact that if only a small number of failures exist in the population, it is more likely that we will sample only a few of them. However, when there are more failures present in the population, then we expect to sample a larger number of failures out of our 30 samples total.

Poisson Distribution and Poisson Process

The *Poisson process* is a *stochastic process* (random process) modeling occurrences of events over time (or more generally in space). This may model the arrival of customers to a system, emission of particles from radioactive material or packets arriving to a communication router. The Poisson process is the canonical example of a *Point process* capturing what may appear as the most sensible

model for completely random occurrences over time. A full description and analysis of the Poisson process is beyond our scope, however we overview the basics.

In a Poisson process, during an infinitesimally small time interval, Δt , it is assumed that (as $\Delta t \rightarrow 0$) there is an occurrence with probability $\lambda\Delta t$ and no occurrence with probability $1 - \lambda\Delta t$. Further, as $\Delta t \rightarrow 0$, it is assumed that the chance of 2 or more occurrences during an interval of length Δt tends to 0. Here $\lambda > 0$ is the *intensity* of the Poisson process, having the property that when multiplied by an interval of length T , the mean number of occurrences during the interval is λT .

The exponential distribution, discussed in the sequel is closely related to the Poisson process as the times between occurrences in the Poisson process are exponentially distributed. Another closely related distribution is the *Poisson distribution* that we discuss now. For a poisson process over the time interval $[0, T]$ the number of occurrences satisfy

$$\mathbb{P}(x \text{ Poisson process occurrences during interval } [0, T]) = e^{-\lambda T} \frac{(\lambda T)^x}{x!} \quad \text{for } x = 0, 1, \dots$$

The Poisson distribution is also sometimes taken as model for occurrences, without explicitly considering a Poisson process. In this case, it often models the number of occurrences during a given time interval. The PMF is then, $p(x) = e^{-\lambda} \lambda^x / x!$, the mean of which is λ . For example, assume that based on previous measurements, on average 5.5 people arrive at a hair salon during rush-hour, then the probability of observing x people during rush-hour can be modeled by the PMF of the Poisson distribution.

As the Poisson process possesses many elegant analytic properties, these sometimes come as an aid when considering Poisson distributed random variables. One such (seemingly magical) property is to consider the random variable $N \geq 0$ such that,

$$\prod_{i=1}^N U_i \geq e^{-\lambda} > \prod_{i=1}^{N+1} U_i \tag{3.12}$$

where U_1, U_2, \dots is a sequence of i.i.d. $\text{uniform}(0, 1)$ random variables and $\prod_{i=1}^0 U_i \equiv 1$. It turns out that seeking such a random variable N produces an efficient recipe for generating a Poisson random variable. That is, the N defined by (3.12) is Poisson distributed with mean λ . Notice that the recipe dictated by (3.12) is to continue multiplying uniform random variables to a “running product” until the product goes below the desired level $e^{-\lambda}$.

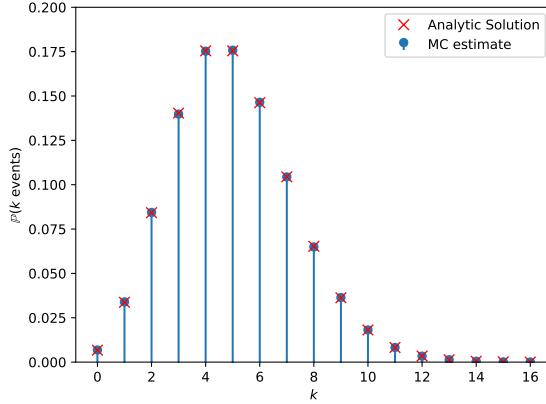
Returning to the hair salon example mentioned above, Listing 3.19 below simulates this scenario, and compares the numerically estimated result against the analytic solution.

Listing 3.19: The Poisson distribution

```

1  using StatsBase, Distributions, PyPlot
2
3  function prn(lambda)
4      k, p = 0, 1
5      while p > MathConstants.e^(-lambda)
6          k += 1
7          p *= rand()
8      end

```

Figure 3.14: The PMF of a Poisson distribution with mean $\lambda = 5.5$.

```

9     return k-1
10    end
11
12 xGrid, lambda, N = 0:16, 5.5, 10^6
13
14 pDist = Poisson(lambda)
15 bPmf = pdf(pDist,xGrid)
16 data = counts([prn(lambda) for _ in 1:N],xGrid)/N
17
18 stem(xGrid,data,label="MC estimate",basefmt="none")
19 plot(xGrid,bPmf,"rx",ms=8,label="PMF")
20 ylim(0,0.2)
21 xlabel(L"$k$")
22 ylabel(L"\mathbb{P}(k \ events)")
23 legend(loc="upper right")

```

- In lines 3-10 we create the function `prn()`, which takes only one argument, the expected arrival rate for our interval `lambda`. It implements (3.12) in a straightforward manner.
- Line 16 calls `prn()`, `N` times, counts occurrences and normalizes by `N` to obtain Monte Carlo estimates of the Poisson probabilities.
- Lines 17-18 plot our Monte Carlo estimates, and the analytic solution of the PMF respectively.

3.6 Families of Continuous Distributions

Like families of discrete distributions, families of continuous distributions are parametrized by a well defined set of parameters. Typically the PDF, $f(x; \theta)$, is parameterized by the *parameter* $\theta \in \Theta$. Hence, technically a family of continuous distributions is the collection of PDFs $f(\cdot; \theta)$ for all $\theta \in \Theta$.

In this section we present some of the most common families of *continuous distributions*. We consider the following: *continuous uniform distribution*, *exponential distribution*, *gamma distribution*, *beta distribution*, *Weibull distribution*, *Gaussian (normal) distribution*, *Rayleigh distribution*

and *Cauchy distribution*. As we did with discrete distributions, the approach that we take in the code examples below often to generate random variables from each such distribution using first principles. We also occasionally dive into related concept that naturally arise in the context of a given distribution. These include the squared coefficient of variation, special functions (gamma and beta), hazard rates, various transformations and heavy tails.

In Listing 3.20 below we illustrate how to create a distribution object for each of the continuous distributions that we investigate in the sequel. The listing is and output style is similar to Listing 3.13 used for discrete distributions.

Listing 3.20: Families of continuous distributions

```

1  using Distributions
2  dists = [
3      Uniform(10,20),
4      Exponential(3.5),
5      Gamma(0.5,7),
6      Beta(10,0.5),
7      Weibull(10,0.5),
8      Normal(20,3.5),
9      Rayleigh(2.4),
10     Cauchy(20,3.5)];
11
12  println("Distribution \t\t\t Parameters \t Support")
13  reshape([dists ; params.(dists) ;
14          ((d)->(minimum(d),maximum(d))).(dists) ],
15          length(dists),3)

```

Continuous Uniform

The *continuous uniform distribution* reflects the case where the outcome of a continuous random variable X has a constant likelihood of occurring over some finite interval. Since the total probability cannot be greater than one, the integral of the PDF (i.e. area under the PDF) must equal one. Therefore given an interval (a, b) , the PDF is given by

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{for } a \leq x \leq b, \\ 0 & \text{for } x < a \text{ or } x > b. \end{cases}$$

As an example, consider the case of a fast spinning circular disk, such as a hard drive. Imagine now there is a small defect on the disk, and we define X as the clockwise angle (in radians) the defect makes with the read head. In this case X is modeled by the continuous uniform distribution over $x \in [0, 2\pi]$. Listing 3.21 below creates Figure 3.15 where we compare between the PDF and a Monte Carlo based estimate.

Listing 3.21: Uniformly distributed angles

```

1  using Distributions, PyPlot
2
3  cUnif = Uniform(0,2*pi)
4  xGrid, N = 0:0.1:2*pi, 10^6

```

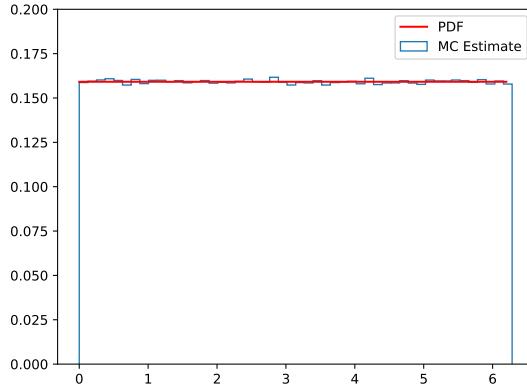


Figure 3.15: The PDF of a continuous uniform distribution over $[0, 2\pi]$.

```

5
6 plt[:hist](rand(N)*2*pi,50, histtype = "step", normed = true,
7      label="MC Estimate")
8 plot(xGrid,pdf(cUnif,xGrid), "r",label="PDF")
9 legend(loc="upper right")
10 ylim(0,0.2)

```

- In line 1 the `Uniform()` function is used to create a continuous uniform distribution over the domain $0, 2\pi$.
- In line 6-7 we simulate N continuous uniform random variables over the domain $[0, 1)$ via the `rand` function, and then scale each of these by a factor of 2π . A histogram of this data is then plotted, using 50 bins total.
- Line 8 uses the `pdf()` function on the distribution object `cUnif` generated in line 3 to plot the analytic PDF. It can be observed that for large N , the numerical estimate is a good approximation of the analytic solution.

Exponential Distribution

As alluded to in the discussion of the Poisson process above, the *exponential distribution* is often used to model random durations between occurrences. A non-negative random variable X , exponentially distributed with rate parameter $\lambda > 0$, has PDF:

$$f(x) = \lambda e^{-\lambda x}.$$

As can be verified, the mean is $1/\lambda$, the variance is $1/\lambda^2$ and the CCDF is $\bar{F}(x) = e^{-\lambda x}$. Note that in Julia, the distribution is parameterized by the mean, rather than by λ . Hence if you wish for say an exponential distribution object with $\lambda = 0.2$ you use `Exponential(5.0)`.

Exponential random variables possess a *lack of memory* property. It can be verified (use the CCDF) that,

$$\mathbb{P}(X > t + s \mid X > t) = \mathbb{P}(X > s).$$

A similar property holds for geometric random variables. This hints at the fact that exponential random variables are the continuous analogs of geometric random variables.

To explore this further, consider a transformation of an exponential random variable X , $Y = \lfloor X \rfloor$ (note the mathematical *floor function* used here). In this case, Y is no longer a continuous random variable, but is discrete in nature, taking on values in the set $\{0, 1, 2, \dots\}$.

We can show that the PMF of Y is,

$$P_Y(y) = \mathbb{P}(\lfloor X \rfloor = y) = \int_y^{y+1} \lambda e^{-\lambda x} dx = (e^{-\lambda})^y (1 - e^{-\lambda}), \quad y = 0, 1, 2, 3, \dots$$

Comparing now to the geometric distribution we set $p = 1 - e^{-\lambda}$, and observe that Y is a geometric random variable (starting at 0) with success parameter p .

In Listing 3.22 below, we show a comparison between the PMF of the floor of an exponential random variable, and the PMF of the geometric distribution covered in Section 3.5. Remember that in Julia, the `Geometric()` has a support that starts at $x = 0$.

Listing 3.22: Flooring an exponential random variable

```

1  using StatsBase, Distributions, PyPlot
2
3  lambda, N = 1, 10^6
4  xGrid = 0:6
5
6  expDist = Exponential(1/lambda)
7  floorData = counts(convert.(Int,floor.(rand(expDist,N))), xGrid)/N
8  geomDist = Geometric(1-MathConstants.e^-lambda)
9
10 stem(xGrid,floorData,label="Floor of Exponential",basefmt="none");
11 plot(xGrid,pdf(geomDist,xGrid), "rx",ms=8,label="Analytic Geometric")
12 ylim(0,1)
13 xlabel("x")
14 ylabel("Probability")
15 legend(loc="upper right");

```

- In lines 3 and 4, we define the value `lambda` of our function, the number of observations we will make `N`, as well as the discrete range of values for which we will plot the PMF.
- In line 6 the `Exponential()` function is used to create the exponential distribution object, `expDist`. Note that the function takes one argument, the inverse of the mean, hence `1/lambda` is used here.
- In line 7 we use the `rand()` function to sample `N` times from the exponential distribution `expDist`. The `floor()` function is then used to round each observation down to the nearest integer, and the `convert()` function used to convert the values from type `Float64` to `Int`. The function `counts()` is then used to count how many times each integer in `xGrid` occurs, and the proportions stored in the array `floorData`.
- In line 8 we use the `Geometric()` function covered previously, to create a geometric distribution object, with probability of success `1-e^-lambda`.

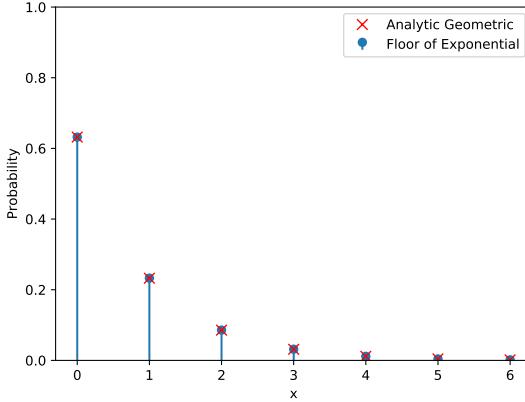


Figure 3.16: The PMF of the floor of an exponential random variable.
It is a geometric distribution.

- Lines 10-15 plot the PMF of both the proportions in `floorData`, and the analytic PMF of the geometric distribution `geomDist`.
- It can be observed that the floor of an exponential random variable is distributed as according to the geometric distribution.

Gamma Distributions and Squared Coefficients of Variation

The *gamma distribution* is a commonly used probability distribution for modeling asymmetric non-negative data. It generalizes the exponential distribution and the chi-squared distribution (covered in Section 5.2, in the context of statistical inference). To introduce this distribution, consider the example where lifetimes of light bulbs are exponentially distributed with mean λ^{-1} . Now imagine we are lighting a room continuously with a single light, and that we replace the bulb with a new one when it burns out. If we start at time 0, what is the distribution of time until n bulbs are replaced?

One way to describe this time, is by the random variable, T where,

$$T = X_1 + X_2 + \dots + X_n,$$

and X_i are i.i.d. exponential random variables (lifetimes of the light bulbs). It turns out that the distribution of T is a gamma distribution.

We now introduce the PDF of the gamma distribution. It is a function (in x) proportional to $x^{\alpha-1}e^{-\lambda x}$, where the non-negative parameters, λ, α are called the *scale parameter* and *shape parameter* respectively. In order to normalize this function we need to divide by,

$$\int_0^\infty x^{\alpha-1} e^{-\lambda x} dx.$$

It turns out that this integral can be represented by $\Gamma(\alpha)/\lambda^\alpha$ where $\Gamma(\cdot)$ is a well known mathematical special function called the *gamma Function*. We investigate the gamma function, and the

related *beta function* and *beta distribution* below. After using the gamma function for normalization, the PDF of the Gamma distribution is,

$$f(x) = \frac{\lambda^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\lambda x}.$$

In the lightbulbs case, we have that $T \sim \text{Gamma}(n, \lambda)$ (with shape parameter $\alpha = n$). In general for a gamma random variable, $Y \sim \text{Gamma}(\alpha, \lambda)$, the shape parameter α does not have to be a whole number. It can analytically be evaluated that,

$$\mathbb{E}[Y] = \frac{\alpha}{\lambda}, \quad \text{and} \quad \text{Var}(Y) = \frac{\alpha}{\lambda^2}.$$

In this respect, it may be interesting to introduce another general notion of variability, often used for non-negative random variables, namely the *squared coefficient of variation*:

$$\text{SCV} = \frac{\text{Var}(Y)}{\mathbb{E}[Y]^2}.$$

The SCV is a normalized (unit-less) version of the variance. The lower it is, the less variability we have in the random variable. It can be seen that for a gamma random variable, the SCV is $1/\alpha$ and for our lightbulb example above, $\text{SCV}(T) = 1/n$. Hence adding more and more lightbulbs, reduces the relative variability.

Listing 3.23 below looks at the three cases of $n = 1$, $n = 10$ and $n = 50$ light bulbs. For each case, we simulate gamma random variables by generating sums of exponential random variables. The resulting histograms are then compared to the theoretical PDF's. Note that the Julia function `Gamma()` is not parametrized by λ , but rather by $1/\lambda$ (i.e. the inverse) in a similar fashion to the `Exponential()` function.

Listing 3.23: Gamma as a sum of exponentials

```

1  using Distributions, PyPlot
2
3  lambda, N = 1/3, 10^5
4  bulbs = [1,10,50]
5  xGrid = 0:0.1:10
6  C = ["b", "r", "g"]
7
8  dists = [Gamma(n, 1/(n*lambda)) for n in bulbs]
9
10 for (n,d) in enumerate(dists)
11     sh = Int64(shape(d))
12     sc = scale(d)
13     data = [sum(-(1/(sh*lambda))*log.(rand(sh))) for _ in 1:N]
14     plt[:hist](data,50,color=C[n], histtype = "step", normed="true")
15     plot(xGrid,pdf(d,xGrid),C[n],
16           label="Shape = $(sh), Scale = $(round(sc,digits=2))")
17 end
18 xlabel("x")
19 ylabel("Probability")
20 xlim(0,maximum(xGrid))
21 legend(loc="upper right");

```

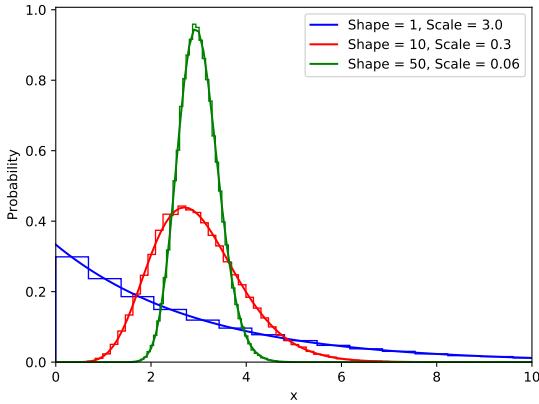


Figure 3.17: Plot of histograms of Monte Carlo simulated gamma observations, against their analytic PDFs.

- In lines 3-6 we specify the main variables of our problem. In line 5 we create the array `bulbs` which stores the number of bulbs in each of our cases. In line 6 we introduce a concept not seen before, where we create an array of type `String`. The characters contained inside this array are used later for color formatting of our plots.
- In line 8 the `Gamma()` function is used along with a comprehension to create a Gamma distribution for each of our cases. The three Gamma distributions are stored in the array `dists`.
- Lines 10-16 contain the core logic of this example, and unlike previous examples, the plotting of the figures is performed inside the `for` loop. The loop is used along with the `enumerate()` function, which creates tuple pairs of index `n` and value (i.e. distribution) `d`, for each distributions in `dists`.
- Line 10 uses the `shape()` and `Int64` functions together to return the shape parameter of each distribution object as an integer.
- Line 12 uses the `scale()` function to return the scale factor of each distribution object.
- Line 13 generates the gamma as a sum of exponential random variables.
- Lines 14 then plots a histogram of our numerically simulated data, and line 15 plots the actual PDF of the distribution via the `pdf()` function. Note that in both cases, the color formatting references the `n`'th index of the array `C`, so that the two separate plots have the same color formatting.

Beta Distribution and Mathematical Special Functions

The *Beta Distribution* is a commonly used distribution when seeking a parameterized shape over a finite support. Namely, beta random variables, parametrized by non-negative, α, β has a density proportional to $x^{\alpha-1}(1-x)^{\beta-1}$ for $x \in [0, 1]$. By using different values of α and β , a variety of shapes can be produced. You may want to try and create such plots yourself to experiment. One common example is $\alpha = 1, \beta = 1$, in which case the distribution defaults to the `uniform(0,1)` distribution.

As with the gamma distribution, we are left to seek a normalizing constant, K such that when multiplied by $x^{\alpha-1}(1-x)^{\beta-1}$, the resulting function has a unit integral over $[0, 1]$. In our case,

$$K = \frac{1}{\int_0^1 x^{\alpha-1}(1-x)^{\beta-1} dx},$$

and hence the PDF is $f(x) = K x^{\alpha-1}(1-x)^{\beta-1}$.

We now explore the beta distribution. By focusing on the normalizing constant, we gain further insight into the Gamma (mathematical) function $\Gamma(\cdot)$, which is a component of the gamma distribution covered above.

It turns out that,

$$K = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)},$$

mathematically, this is called the inverse of the *Beta function*, evaluated at α and β . Let us focus solely on the gamma functions, with the purpose of demystifying their use in the gamma and beta distributions. The mathematical function `gamma` is a type of *special function*, and is defined as,

$$\Gamma(z) = \int_0^\infty x^{z-1} e^{-x} dx.$$

It turns out that it is a continuous generalization of factorial. We know that for positive integer n ,

$$n! = n * (n - 1)!, \quad \text{with } 0! \equiv 1.$$

This is the recursive definition of factorial. The gamma function exhibits similar properties, as one can evaluate it via integration by parts,

$$\Gamma(z) = (z - 1) \Gamma(z - 1).$$

Note further that, $\Gamma(1) = 1$. Hence we see that for integer values of z ,

$$\Gamma(z) = (z - 1)!.$$

We now illustrate this in Listing 3.24 below, and in the process take into consideration the mathematical function `beta` and the beta PDF. Observe the difference in Julia between case-sensitive `gamma()` (the special mathematical function) and `Gamma()` (the constructor for the distribution). Similarly for `beta()` and `Beta()`.

Listing 3.24: The gamma and beta special functions

```

1  using SpecialFunctions, Distributions
2
3  a,b = 0.2, 0.7
4
5  betaAB1 = beta(a,b)
6  betaAB2 = (gamma(a)gamma(b))/gamma(a+b)
7  betaAB3 = (factorial(a-1)factorial(b-1))/factorial(a+b-1)
8  betaPDFAB1 = pdf(Beta(a,b),0.75)
9  betaPDFAB2 = (1/beta(a,b))*0.75^(a-1) * (1-0.75)^(b-1)
10
11 println("beta($a,$b)      = $betaAB1,\t$betaAB2,\t$betaAB3 ")
12 println("betaPDF($a,$b) = $betaPDFAB1,\t$betaPDFAB2")

```

```
beta(0.2, 0.7)      = 5.576463695849875,      5.576463695849875,      5.576463695849877
betaPDF(0.2, 0.7)   = 0.34214492891381176,  0.34214492891381176
```

Another important property of the gamma function that we encounter later on (in the context of the Chi squared distribution, which we touch on in Section 5.2) is $\Gamma(1/2) = \sqrt{\pi}$. We show this now through numerical integration in Listing 3.25 below.

Listing 3.25: The gamma function at 1/2

```
1  using QuadGK
2
3  g(x) = x^(0.5-1) * MathConstants.e^-x
4  quadgk(g, 0, Inf)[1], sqrt(pi)
```

```
(1.7724538355037913, 1.7724538509055159)
```

- This example uses the `QuadGK` package, in the same manner as introduced in Listing 3.3. We can see that the numerical integration is in agreement with the analytically expected result.

Weibull Distribution and Hazard Rates

We now look at the *Weibull Distribution* and explore the concept of the *hazard rate function*, which is often used in reliability and survival analysis. For random variables T , indicating the lifetime of an individual or a component, an interesting quantity is the instantaneous chance of death at any time, given that the individual has lived until now. Mathematically this is,

$$h(x) = \lim_{\Delta \rightarrow 0} \frac{1}{\Delta} \mathbb{P}(T \in [x, x + \Delta) \mid T > x),$$

or alternatively (by using the conditional probability and noticing that the PDF $f(x)$ satisfies $f(x)\Delta \approx \mathbb{P}(x \leq T < x + \Delta)$ for small Δ),

$$h(x) = \frac{f(x)}{1 - F(x)}.$$

The function $h(\cdot)$ is called the hazard rate, and is the common method of viewing the distribution for lifetime random variables, T . In fact, we can reconstruct the CDF, $F(x)$ by,

$$1 - F(x) = \exp \left(- \int_0^x h(t) dt \right).$$

The *Weibull distribution* is naturally defined through the hazard rate. It is a distribution with,

$$h(x) = \lambda x^\alpha.$$

Notice that the parameter α gives the Weibull distribution different modes of behavior. If $\alpha = 0$ then the hazard rate is constant, in which case the Weibull distribution is actually an exponential distribution with rate λ . If on the other hand $\alpha > 0$, then the hazard rate increases over time. This depicts a situation of “aging components”, i.e. the longer a component has lived, the higher the instantaneous chance of failure. It is sometimes called *Increasing Failure Rate (IFR)*. Conversely,

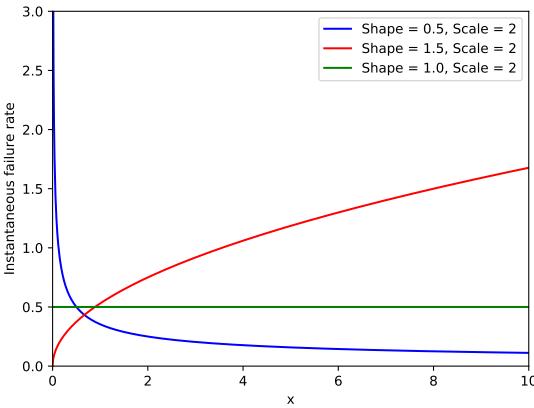


Figure 3.18: Hazard rate functions for different Weibull distributions.

$\alpha < 0$ depicts a situation where the longer a component has lasted, the lower the chance of it failing (as is perhaps the case with totalitarian political regimes). It is sometimes called *Decreasing Failure Rate (DFR)*.

In Listing 3.26 below, we look at several hazard rate functions for different Weibull distributions.

Listing 3.26: Hazard rates and the Weibull distribution

```

1  using Distributions, PyPlot
2
3  A = [0.5 1.5 1]
4  dists = [Weibull(a,2) for a in A]
5  xGrid = 0:0.01:10
6  C = ["b", "r", "g"]
7
8  for (n,d) in enumerate(dists)
9      plot(xGrid,pdf(d,xGrid)./(ccdf(d,xGrid)),C[n],
10         label="Shape = $(shape(d)), Scale = 2")
11  end
12  xlabel("x")
13  ylabel("Instantaneous failure rate")
14  xlim(0,10)
15  ylim(0,3)
16  legend(loc="upper right")

```

- In line 3, we store the different shape factors we will be considering in the array A.
- Line 4 uses the `Weibull()` function along with a comprehension to create several Weibull distributions, with the shape factors specified in A (note for each, we use the same scale factor 2).
- In lines 8-11 the hazard rate for each Weibull distribution is plotted. Line 9 implements the formula for the hazard rate, which was defined above. Note the use of the `pdf()` and the `ccdf()` functions in the calculation.

Gaussian (Normal) Distribution

Arguably, the most well known distribution is the *normal distribution*, also known as the *Gaussian distribution*. It is a symmetric “bell curved” shaped distribution, which can be found throughout nature. Examples include the distribution of heights among adult humans, and the distribution of IQ’s. It is an important distribution, and is commonly found due to a property called the central limit theorem, which is covered in more depth in Section 5.3.

The Gaussian distribution is defined by two parameters, μ and σ^2 , which are the mean and variance respectively. The phrase *standard normal* signifies the case of a normal distribution with $\mu = 0$ and $\sigma^2 = 1$. The PDF of the normal distribution is given by,

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

As an illustrative example, we plot the normal PDF, along with its first and second derivatives in Listing 3.27 below. The first derivative is clearly 0 at the PDF’s unique maximum at $x = 0$. The second derivative is 0 at the points $x = -1$ and $x = +1$. These are exactly the *inflection points* of the normal PDF.

Listing 3.27: Numerical derivatives of the normal density

```

1  using Distributions, Calculus ,PyPlot
2
3  xGrid = -5:0.01:5
4  sig = 1.5
5  normalDensity(z) = pdf(Normal(0,sig),z)
6
7  d0 = normalDensity.(xGrid)
8  d1 = derivative.(normalDensity,xGrid)
9  d2 = second_derivative.(normalDensity, xGrid)
10
11 ax = gca()
12 plot(xGrid,d0,"b",label="f(x)")
13 plot(xGrid,d1,"r",label="f'(x)")
14 plot(xGrid,d2,"g",label="f''(x)")
15 plot([-5,5],[0,0],"k", lw=0.5)
16 xlabel("x")
17 xlim(-5,5)
18 legend(loc="upper right")
```

- In this example we use the functions `derivative` and `second_derivative`, which both come from the `Calculus` package.
- In line 5 we define the function `normalDensity`, which takes an input z , and returns the corresponding value of the PDF of a normal distribution, centered at 0, with a standard deviation of `sig`.
- In lines 7-8, the functions `derivative()` and `second_derivative()` are used to evaluate the first and second derivatives of `normalDensity` respectively.

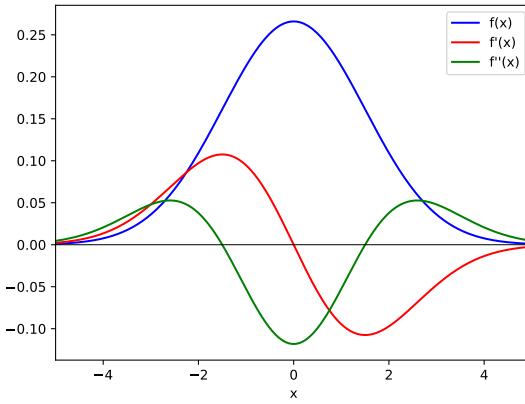


Figure 3.19: Plot of the standard normal PDF and its first and second derivatives.

Rayleigh Distribution and the Box-Muller Transform

We now consider an exponentially distributed random variable, X , with rate parameter $\lambda = \sigma^{-2}/2$ where $\sigma > 0$. If we set a new random variable, $R = \sqrt{X}$, what is the distribution of R ? To work this out analytically, we have for $y \geq 0$,

$$F_R(y) = \mathbb{P}(\sqrt{X} \leq y) = \mathbb{P}(X \leq y^2) = F_X(y^2) = 1 - \exp\left(-\frac{y^2}{2\sigma^2}\right),$$

and by differentiating, we get the density,

$$f_R(y) = \frac{y}{\sigma^2} \exp\left(-\frac{y^2}{2\sigma^2}\right).$$

This is the density of the *Rayleigh Distribution* with parameter σ . We see it is related to the exponential distribution via a square root transformation. Hence the implication is that since we know how generate exponential random variables via $-\frac{1}{\lambda} \log(U)$ where $U \sim \text{uniform}(0, 1)$, then if we take the square root of that we can generate Rayleigh random variables.

The Rayleigh distribution is important because of another distributional relationship. Consider two independent normally distributed random variables, N_1 and N_2 , each with mean 0 and standard deviation σ . In this case, it turns out that $\tilde{R} = \sqrt{N_1^2 + N_2^2}$ is Rayleigh distributed just as R above. As we see in the sequel this property yields a method for generating normal random variables. It also yields a statistical model often used in radio communications called *Rayleigh fading*.

The code listing below demonstrates three alternative ways of generating Rayleigh random variables. It generates R as above, \tilde{R} as above and uses the `rand()` command applied to a Rayleigh object from the `Distributions` package. The mean of a Rayleigh random variable is $\sigma\sqrt{\pi/2}$ and is approximately 2.1306 when $\sigma = 1.7$, as in the code below.

Listing 3.28: Alternative representations of Rayleigh random variables

```
1  using Distributions
2
```

```

3 N = 10^6
4 sig = 1.7
5
6 data1 = sqrt.(-(2* sig^2)*log.(rand(N)))
7
8 distG = Normal(0,sig)
9 data2 = sqrt.(rand(distG,N).^2 + rand(distG,N).^2)
10
11 distR = Rayleigh(sig)
12 data3 = rand(distR,N)
13
14 mean.([data1, data2, data3])

```

```

3-element Array{Float64,1}:
 2.12994
 2.12935
 2.13188

```

- Line 6 generates `data1`, as in R above. Note the use of element wise mapping of `sqrt` and `log`.
- Lines 8 and 9 generate `data2`, as in \tilde{R} above. Here we use `rand()` applied to `distG`, a Normal distribution object from the `Distributions` package.
- Lines 11 and 12 use `rand()` applied to a Rayleigh distribution object. The implementation is most probably as in line 9.
- Line 14 produces the output by applying `mean` to `data1`, `data2` and `data3` individually. Observe that the sample mean is very similar to the theoretical mean presented above.

A common way to generate normal random variables, called the *Box-Muller Transform*, is to use the relationship between the Rayleigh distribution and a pair of independent zero mean Normal random variables, as mentioned above. Consider Figure 3.20 representing the relationship between the pair (N_1, N_2) and their *polar coordinate* counterparts, R and θ . Assume now that the Cartesian coordinates of the point, (N_1, N_2) are identically normally distributed, with N_1 independent of N_2 . In this case, by representing N_1 and N_2 in polar coordinates (θ, R) we have that the angle, θ is uniformly distributed on $[0, 2\pi]$ and that the radius R is distributed as a Rayleigh random variable. Hence a recipe for generating N_1 and N_2 is to generate θ and R and transform via,

$$N_1 = R \cos(\theta), \quad N_2 = R \sin(\theta).$$

Often, N_2 is not even needed. Hence in practice, given two independent uniform(0,1) random variables, U_1 and U_2 we set,

$$Z = \sqrt{-2 \ln U_1} \cos(2\pi U_2).$$

and it has a standard Normal distribution. Listing 3.29 uses this method to generate Normal random variables and compares their histogram to the standard normal PDF. The output is in Figure 3.21.

Listing 3.29: The Box-Muller transform

```

1 using Random, Distributions, PyPlot
2 Random.seed!(1)

```

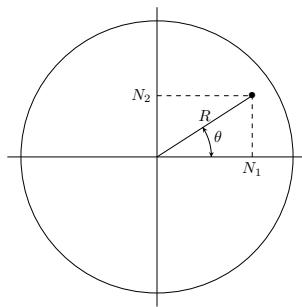


Figure 3.20: Geometry of the Box-Muller transform.

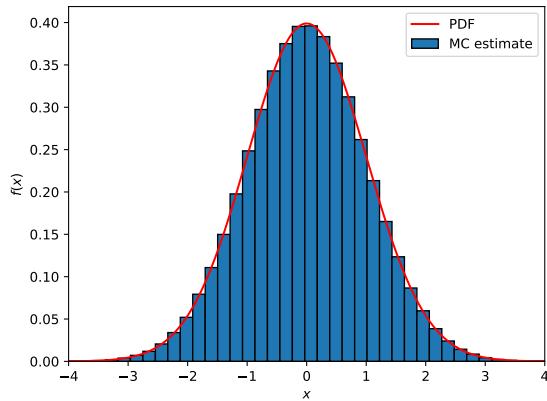


Figure 3.21: The Box-Muller transform can be used to generate a normally distributed random variable.

```

3
4 Z() = sqrt(-2*log(rand()))*cos(2*pi*rand())
5 xGrid = -4:0.01:4
6
7 plt[:hist]([Z() for _ in 1:10^6],50,
8     normed = true,ec="k",label="MC estimate")
9 plot(xGrid, pdf(Normal(),xGrid), "-r",label="PDF")
10 xlim(-4,4)
11 xlabel(L"$x$")
12 ylabel(L"$f(x)$")
13 legend(loc="upper right")

```

- In line 4 we define a function, `X()` which implements the Box-Muller transform.
- In lines 7-8, the plotting of the histogram specifies 50 as the number of bins.
- Notice the `xlabel()` and `ylabel()` functions in lines 11-12 use the `L` for latex formatting.

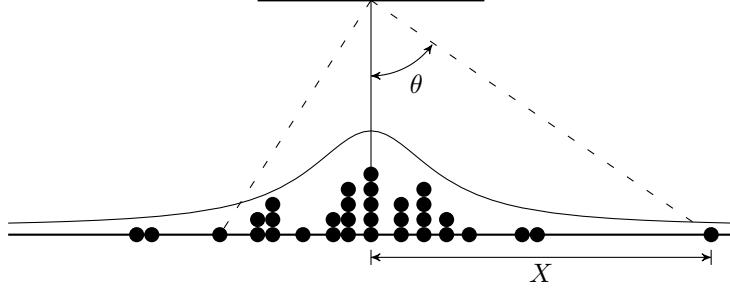


Figure 3.22: Indicative distribution of laser shots from a hovering drone.

Cauchy Distribution

At first glance, the PDF of the *Cauchy distribution* (also known as the *Lorentz distribution*) looks very similar to the normal distribution. However, it is fundamentally different, as its mean and standard deviation are undefined. The PDF of the Cauchy distribution is given by,

$$f(x) = \frac{1}{\pi\gamma \left(1 + \left(\frac{x - x_0}{\gamma}\right)^2\right)} \quad (3.13)$$

where x_0 is the location parameter at which the peak is observed and γ is the scale parameter.

In order to understand the context of this type of distribution we will develop a real-world example of a Cauchy distributed random variable. Consider a drone hovering stationary in the sky at unit height. A pivoting laser is attached to the undercarriage, which pivots back and forth as it shoots pulses at the ground. At any point the laser fires, it makes an angle θ from the vertical ($-\pi/2 \leq \theta \leq \pi/2$). This is illustrated in Figure 3.22.

Since the laser fires at a high frequency as it is pivoting, we can assume that the angle θ is distributed uniformly on $[-\pi/2, \pi/2]$. For each shot from the laser, a point can be measured, X , horizontally on the ground from the point above which the drone is hovering. Hence we can now consider this horizontal measurement as a new random variable, X . Now,

$$F_X(x) = \mathbb{P}(X \leq x) = \mathbb{P}(\tan(\theta) \leq x) = \mathbb{P}(\theta \leq \arctan(x)) = F_\theta(\arctan(x)) = \begin{cases} 0 & x < -\pi/2, \\ \frac{1}{\pi} \arctan(x) & x \in [-\pi/2, \pi/2], \\ 1 & \pi/2 < x. \end{cases}$$

The density is obtained by taking the derivative, which evaluates to,

$$f(x) = \frac{1}{\pi(1+x^2)}.$$

This is a special case ($x_0 = 0$ and $\gamma = 1$) of the more complicated density (3.13). Now the integral,

$$\int_{-\infty}^{\infty} xf(x) dx,$$

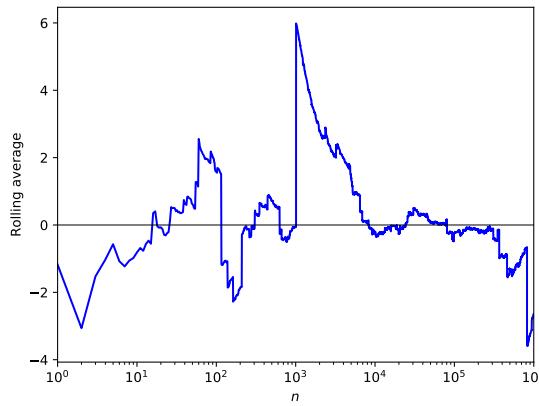


Figure 3.23: Cumulative average of Cauchy distributed random variables.

is not defined since each of the one sided improper integrals does not converge. Hence a Cauchy random variable is an example of a *distribution without a mean*. You may now ask, what happens to sample averages of such random variables. That is, would the sequence of sample averages converge to anything? The answer is no. We illustrate this in the listing below and the accompanying Figure 3.23.

Listing 3.30: The law of large numbers breaks down with very heavy tails

```

1  using Random,PyPlot
2  Random.seed!(808)
3
4  n = 10^6
5  data = tan.(rand(n)*pi .- pi/2)
6  averages = accumulate(+,data)./collect(1:n)
7
8  plot(1:n,averages,"b")
9  plot([1,n],[0,0],"k",lw=0.5)
10 xscale("log")
11 xlim(0,n)
12 xlabel(L"$n$")
13 ylabel("Rolling average")

```

- In line 2 the seed of the random number generator is set, so that the same stream of random numbers is generated each time.
- In line 5 we create `data`, an array of `n` Cauchy random variables. The construction is directly through the angle mechanism described above (Figure 3.22).
- In line 6 we use the `accumulate()` function to create a running sum and then divide (element wise via `.` by the array, `collect(1:n)`). Notice that '+' is used as a first argument to `accumulate()`. Here the addition operator is actually treated as a function.
- The remainder of the code plots the running average. As is apparent from Figure 3.23, occasional large values (due to angles near $-\pi/2$ or $\pi/2$) create huge spikes. There is no strong law of large numbers in this case (since the mean is not defined).

3.7 Joint Distributions and Covariance

We now consider pairs and vectors of random variables. In general, in a probability space, we may define multiple random variables, X_1, \dots, X_n where we consider the vector or tuple, $\mathbf{X} = (X_1, \dots, X_n)$ as a *random vector*. A key question deals with representing and evaluating probabilities of the form $\mathbb{P}(\mathbf{X} \in B)$, where B is some subset \mathbb{R}^n . Our focus here is on the case of a pair of random variables, denoted (X, Y) where they are continuous and have a density function. The probability distribution of (X, Y) is called a *bivariate distribution* and more generally, the probability distribution of \mathbf{X} is called a *multi-variate distribution*.

The Joint PDF

A function, $f_{\mathbf{X}} : \mathbb{R}^n \rightarrow \mathbb{R}$ is said to be a *joint probability density function* (PDF) if

$$(1) \quad f_{\mathbf{X}}(x_1, x_2, \dots, x_n) \geq 0.$$

$$(2) \quad \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} f_{\mathbf{X}}(x_1, x_2, \dots, x_n) dx_1 dx_2 \dots dx_n = 1.$$

Considering now $B \subset \mathbb{R}^n$, probabilities of a random vector, \mathbf{X} distributed with density $f_{\mathbf{X}}$, can be evaluated via,

$$\mathbb{P}(\mathbf{X} \in B) = \int_B f_{\mathbf{X}}(\mathbf{x}) d\mathbf{x}.$$

As an example let $\mathbf{X} = (X, Y)$ and consider the joint density,

$$f(x, y) = \begin{cases} \frac{9}{8}(4x + y)\sqrt{(1-x)(1-y)}, & x \in [0, 1], y \in [0, 1], \\ 0, & \text{otherwise.} \end{cases}$$

This PDF is plotted in Figure 3.24. We may now obtain all kinds of probabilities for example, set $B = \{(x, y) \mid x + y > 1\}$, then,

$$\mathbb{P}((x, y) \in B) = \int_{x=0}^1 \int_{y=x}^1 f(x, y) dy dx = \frac{31}{80} = 0.3875.$$

The joint distribution of X and Y allows us to also obtain related distributions. We may obtain the *marginal densities* of X and Y , denoted $f_X(\cdot)$ and $f_Y(\cdot)$, via,

$$f_X(x) = \int_{y=0}^1 f(x, y) dy, \quad \text{and} \quad f_Y(y) = \int_{x=0}^1 f(x, y) dx.$$

For our example by explicitly integrating we obtain,

$$f_X(x) = \frac{3}{10}\sqrt{1-x}(1+10x) \quad \text{and} \quad f_Y(y) = \frac{3}{20}\sqrt{1-y}(8+5y).$$

In general, the random variables X and Y are said to be *independent* if, $f(x, y) = f_X(x)f_Y(y)$. In our current example, this is not the case. Further, whenever we have two densities of scalar random

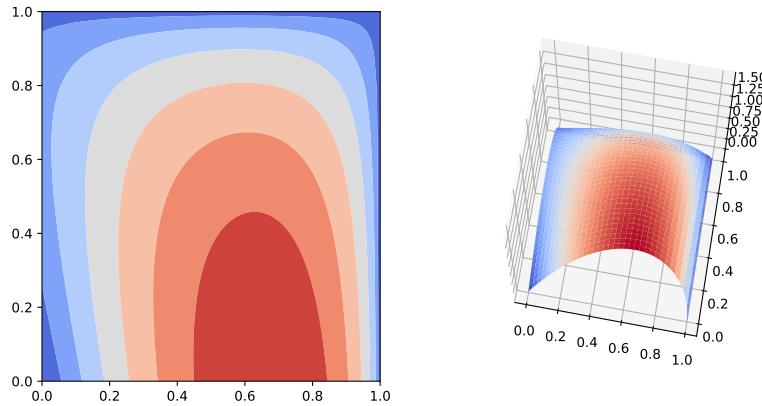


Figure 3.24: A contour plot and a three dimensional surface plot of $f(x, y)$.

variables, we may multiply them to make the joint distribution of the random vector composed of independent copies. That is, if we take our $f_X(\cdot)$ and $f_Y(\cdot)$ above, we may create, $\tilde{f}(x, y)$ via,

$$\tilde{f}(x, y) = f_X(x)f_Y(y) = \frac{9}{200}\sqrt{1-x}(10x+1)\sqrt{1-y}(5y+8).$$

Observe that $\tilde{f}(x, y) \neq f(x, y)$. Hence we see, that while both bivariate distributions have the same marginal distribution, they are different bivariate distributions and hence describe different relationships between X and Y .

Of further interest is the *conditional density* of X given Y (and vice-versa). It is denoted by $f_{X|Y=y}(x)$ and describes the distribution of the random variable X , given the specific value $Y = y$. It can be obtained from the joint density via,

$$f_{X|Y=y}(x) = \frac{f(x, y)}{f_Y(y)} = \frac{f(x, y)}{\int_{x=0}^1 f(x, y) dx}.$$

The code below generates Figure 3.24.

Listing 3.31: Visualizing a bivariate density

```

1  using PyPlot
2
3  support = 0:0.01:1
4  f(x,y) = 9/8*(4x+y)*sqrt((1-x)*(1-y))
5
6  z = [ f(j,i) for i in support, j in support]
7
8  fig = figure(figsize=(10,5))
9  ax1 = fig[:add_subplot](121)
10  contourf(support, support, z, cmap="coolwarm")
11
12  ax2 = fig[:add_subplot](122, projection = "3d")
13  ax2[:view_init](elev=70, azim=-80)
14  plot_surface(support, support, z, cmap="coolwarm")

```

- In line 3 we define the support on which to plot (for a single dimension).
- In line 4 we define the bivariate density function, $f()$.
- Line 6 is a comprehension with two running indices, i and j . The type of z is the `Array{Float64, 2}`. The remainder of the code plots this surface.
- In lines 8-10 we create a contour plot of it.
- In lines 12-14 we create a surface plot of it. Notice the elevation and azimuth parameters specified in line 13.

Covariance and Vectorized Moments

Given two random variables, X and Y , with respective means, μ_X and μ_Y , the *covariance* is defined by,

$$\text{Cov}(X, Y) = \mathbb{E}[(X - \mu_X)(Y - \mu_Y)] = \mathbb{E}[XY] - \mu_x\mu_y.$$

The second formula follows by expansion. Notice also that $\text{Cov}(X, X) = \text{Var}(X)$. Compare with formula (3.3). The covariance is a common measure of the relationship between the two random variables where if $\text{Cov}(X, Y) = 0$ we say the random variables are *uncorrelated*. Further, if $\text{Cov}(X, Y) \neq 0$, the sign of it gives an indication of the relationship.

The *correlation coefficient*, denoted ρ_{XY} , is

$$\rho_{XY} = \frac{\text{cov}(X, Y)}{\sqrt{\text{Var}(X)\text{Var}(Y)}}. \quad (3.14)$$

It is a normalized form of the covariance with,

$$-1 \leq \rho_{XY} \leq 1.$$

Values nearing ± 1 indicate a very strong *linear relationship* between X and Y and values near or at 0 indicate a lack of a linear relationship.

Note that if X and Y are independent random variables then $\text{Cov}(X, Y) = 0$ and $\rho_{XY} = 0$. The opposite case does not always hold: In general $\rho_{XY} = 0$ does not imply independence. However as described below, for jointly Normal random variables it does.

Consider now a random vector $\mathbf{X} = (X_1, \dots, X_n)$ (taken as a column vector). It can be described by moments in an analogous manner to a scalar random variable, see Section 3.2. A key quantity is the *mean vector*,

$$\mu_X := [\mathbb{E}[X_1], \mathbb{E}[X_2], \dots, \mathbb{E}[X_n]]'.$$

Further, the *covariance matrix* is the matrix defined by the expectation (taken element wise) of the (*outer product*) random matrix given by $(X - \mu_X)(X - \mu_X)'$, and is expressed as

$$\Sigma_X = \text{Cov}(X) = \mathbb{E}[(X - \mu_X)(X - \mu_X)']. \quad (3.15)$$

As can be verified, the i, j 'th element of Σ_X is $\text{Cov}(X_i, X_j)$ and hence the diagonal elements are the variances.

Linear Combinations and Transformations

For any collection of random variables,

$$\mathbb{E}[X_1 + \dots + X_n] = \mathbb{E}[X_1] + \dots + \mathbb{E}[X_n].$$

For uncorrelated random variables,

$$\text{Var}(X_1 + \dots + X_n) = \text{Var}(X_1) + \dots + \text{Var}(X_n).$$

More generally (allowing the random variables to be correlated),

$$\text{Var}(X_1 + \dots + X_n) = \text{Var}(X_1) + \dots + \text{Var}(X_n) + 2 \sum_{i < j} \text{Cov}(X_i, X_j). \quad (3.16)$$

Note that the right hand side of (3.16) is the sum of the elements of the matrix $\text{Cov}((X_1, \dots, X_n))$. This is a special case of a more general *affine transformation* where we take a random vector $\mathbf{X} = (X_1, \dots, X_n)$ with covariance matrix $\Sigma_{\mathbf{X}}$, and an $m \times n$ matrix A and m vector \mathbf{b} . We then set

$$\mathbf{Y} = A\mathbf{X} + \mathbf{b}. \quad (3.17)$$

In this case, the new random vector \mathbf{Y} exhibits mean and covariance,

$$\mathbb{E}[\mathbf{Y}] = A\mathbb{E}[\mathbf{X}] + \mathbf{b} \quad \text{and} \quad \text{Cov}(\mathbf{Y}) = A\Sigma_{\mathbf{X}}A', \quad (3.18)$$

Now to retrieve (3.16), we use the $1 \times n$ matrix $A = [1, \dots, 1]$ and observe that $A'\Sigma_{\mathbf{X}}A$ is a sum of all of the elements of $\Sigma_{\mathbf{X}}$.

The Cholesky Decomposition and Generating Random Vectors

Say now that you wish to create an n dimensional random vector \mathbf{Y} with some specified mean vector $\mu_{\mathbf{Y}}$ and covariance matrix $\Sigma_{\mathbf{Y}}$. That is, $\mu_{\mathbf{Y}}$ and $\Sigma_{\mathbf{Y}}$ are known.

The formulas (3.18) yield a potential recipe for such a task if we are given a random vector \mathbf{X} with zero mean and identity covariance matrix ($\Sigma_{\mathbf{X}} = I$). For example in the context of Monte Carlo random variable generation, creating such a random vector \mathbf{X} is trivial – just generate a sequence of n i.i.d. $\text{Normal}(0,1)$ random variables.

Now apply the affine transformation (3.17) on \mathbf{X} with $\mathbf{b} = \mu_{\mathbf{Y}}$ and a matrix A that satisfies,

$$\Sigma_{\mathbf{Y}} = AA'. \quad (3.19)$$

Now (3.18) guarantees that \mathbf{Y} has the desired $\mu_{\mathbf{Y}}$ and $\Sigma_{\mathbf{Y}}$.

The question is now how to find a matrix A that satisfies (3.19). For this the *Cholesky decomposition* comes as an aid. As an example assume we wish to generate a random vector \mathbf{Y} with,

$$\mu_{\mathbf{Y}} = \begin{bmatrix} 15 \\ 20 \end{bmatrix} \quad \text{and} \quad \Sigma_{\mathbf{Y}} = \begin{bmatrix} 9 & 4 \\ 4 & 16 \end{bmatrix}.$$

The code below generates random vectors with these mean vector and covariance matrix using three alternative forms of zero-mean, identity-covariance matrix random variables. As you can see from Figure 3.25, such distributions can be very different in nature even though they share the same first and second order characteristics. The output also presents mean and variance estimates of the generated random variables showing they agree with the specifications above.

Listing 3.32: Generating random vectors with desired mean and covariance

```

1  using Distributions, LinearAlgebra, PyPlot
2
3  SigY = [ 6 4 ; 4 9]
4  muY = [15 ; 20]
5  A = cholesky(SigY).L
6
7  N = 10^5
8
9  dist_a = Normal()
10 rvX_a() = [rand(dist_a) ; rand(dist_a)]
11 rvY_a() = A*rvX_a() + muY
12 data_a = [rvY_a() for _ in 1:N]
13 data_a1 = first.(data_a)
14 data_a2 = last.(data_a)
15
16 dist_b = Uniform(-sqrt(3),sqrt(3))
17 rvX_b() = [rand(dist_b) ; rand(dist_b)]
18 rvY_b() = A*rvX_b() + muY
19 data_b = [rvY_b() for _ in 1:N]
20 data_b1 = first.(data_b)
21 data_b2 = last.(data_b)
22
23 dist_c = Exponential()
24 rvX_c() = [rand(dist_c) - 1; rand(dist_c) - 1]
25 rvY_c() = A*rvX_c() + muY
26 data_c = [rvY_c() for _ in 1:N]
27 data_c1 = first.(data_c)
28 data_c2 = last.(data_c)
29
30 plot(data_a1,data_a2,".",color="blue",ms=0.2);
31 plot(data_b1,data_b2,".",color="red",ms=0.2);
32 plot(data_c1,data_c2,".",color="green",ms=0.2);
33
34 stats(data1,data2) = println(
35     round(mean(data1),digits=2), "\t",
36     round(mean(data2),digits=2), "\t",
37     round(var(data1),digits=2), "\t",
38     round(var(data2),digits=2), "\t",
39     round(cov(data1,data2),digits=2))
40
41 println("Mean1\tMean2\tVar1\tVar2\tCov")
42 stats(data_a1,data_a2)
43 stats(data_b1,data_b2)
44 stats(data_c1,data_c2)
```

Mean1	Mean2	Var1	Var2	Cov
14.99	20.0	6.02	8.97	3.97
15.0	20.0	5.99	8.98	3.96
15.0	20.01	6.07	9.05	4.06

- In line 5 the `cholesky()` function from the `LinearAlgebra` package is used on `SigY` to

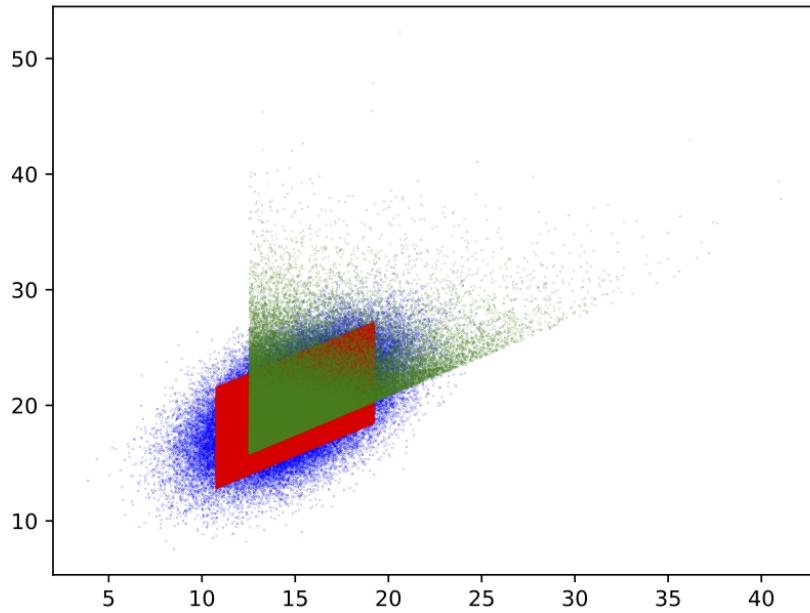


Figure 3.25: Random vectors from three different distributions, each sharing the same mean and covariance matrix.

find a lower triangular matrix that is a Cholesky decomposition of it. Note the use of the `.L`, which returns the `L` field of the result of `cholesky()`. We recommend referring to the help for `cholfact()` for more details.

- The code then has three distinct blocks, each utilizing a different basic univariate distribution. In lines 9-14 we use a `Normal()` distribution (standard normal). In lines 16-21, we use a uniform distribution over $[-\sqrt{3}, \sqrt{3}]$. This distribution has zero mean and unit variance. In lines 23-28, we use an exponential distribution with $\lambda = 1$, shifted by -1 so that it has zero mean and unit variance.
- In lines 30-32 we plot the data points for these random vectors.
- In line 34-37 we define the function `stats()`, which prints the sample mean and sample variance of the input argument arrays. The remainder of the code prints the output seen below the listing.

Bivariate Normal

One of the most ubiquitous families of multi-variate distributions is the *multi-variate normal distribution*. Similarly to the fact that a scalar (*univariate*) normal distribution is parametrized by the mean μ and the variance σ^2 , a multi-variate normal distribution is parametrized by the mean vector $\mu_{\mathbf{X}}$ and the covariance matrix $\Sigma_{\mathbf{X}}$.

Begin first with the *standard multi-variate* having $\mu_{\mathbf{X}} = 0$ mean and $\Sigma_{\mathbf{X}} = I$. In this case, the PDF for the random vector $\mathbf{X} = (X_1, \dots, X_n)$ is,

$$f(\mathbf{x}) = (2\pi)^{-n/2} e^{-\frac{1}{2}\mathbf{x}'\mathbf{x}}.$$

The example below illustrates numerically that this is a valid PDF for $n = 2$ via numerical integration.

Listing 3.33: Multidimensional integration

```

1  using HCubature
2  M = 4
3  f(x) = (2*pi)^(-length(x)/2) * exp(-(1/2)*x'*x)
4  hcubature(f, [-M, M], [-M, M])

```

Now in general, using an affine transformation like (3.17), it can be shown that for arbitrary $\mu_{\mathbf{X}}$ and $\Sigma_{\mathbf{X}}$,

$$f(\mathbf{x}) = |\Sigma_{\mathbf{X}}|^{-1/2} (2\pi)^{-n/2} e^{-\frac{1}{2}(\mathbf{x}-\mu_{\mathbf{X}})' \Sigma_{\mathbf{X}}^{-1} (\mathbf{x}-\mu_{\mathbf{X}})},$$

where $|\cdot|$ is the determinant. In the case of $n = 2$, this becomes the *bivariate normal distribution* with a density represented as,

$$f_{XY}(x, y; \sigma_X, \sigma_Y, \mu_X, \mu_Y, \rho) = \frac{1}{2\pi\sigma_X\sigma_Y\sqrt{1-\rho^2}} \times \exp\left\{ \frac{-1}{2(1-\rho^2)} \left[\frac{(x-\mu_X)^2}{\sigma_X^2} - \frac{2\rho(x-\mu_X)(y-\mu_Y)}{\sigma_X\sigma_Y} + \frac{(y-\mu_Y)^2}{\sigma_Y^2} \right] \right\}$$

Here the elements of the mean and covariance matrix are spelled out via,

$$\mu_{\mathbf{X}} = \begin{bmatrix} \mu_X \\ \mu_Y \end{bmatrix} \quad \text{and} \quad \Sigma_{\mathbf{Y}} = \begin{bmatrix} \sigma_X^2 & \sigma_X\sigma_Y\rho \\ \sigma_X\sigma_Y\rho & \sigma_Y^2 \end{bmatrix}.$$

Note that $\rho \in [-1, 1]$ is the correlation coefficient as defined in (8.13).

In Section 4.2 we fit the five parameters of a bivariate normal to weather data. The example below, illustrates a plot of random vectors generated from a distribution matching these parameters.

Listing 3.34: Bivariate normal data

```

1  using Distributions, PyPlot
2
3  include("mvParams.jl")
4  biNorm = MvNormal(meanVect, covMat)
5
6  N = 10^3
7  points = rand(MvNormal(meanVect, covMat), N)
8
9  support = 15:0.1:40
10 Z = [ pdf(biNorm, [i, j]) for i in support, j in support ]
11
12 fig = figure(figsize=(10, 5))
13 ax = fig[:add_subplot](121)
14 plot(points[1, :], points[2, :], ".", ms=1, label="scatter")
15 CS = contour(support, support, copy(Z'), levels=[0.001, 0.005, 0.02])

```

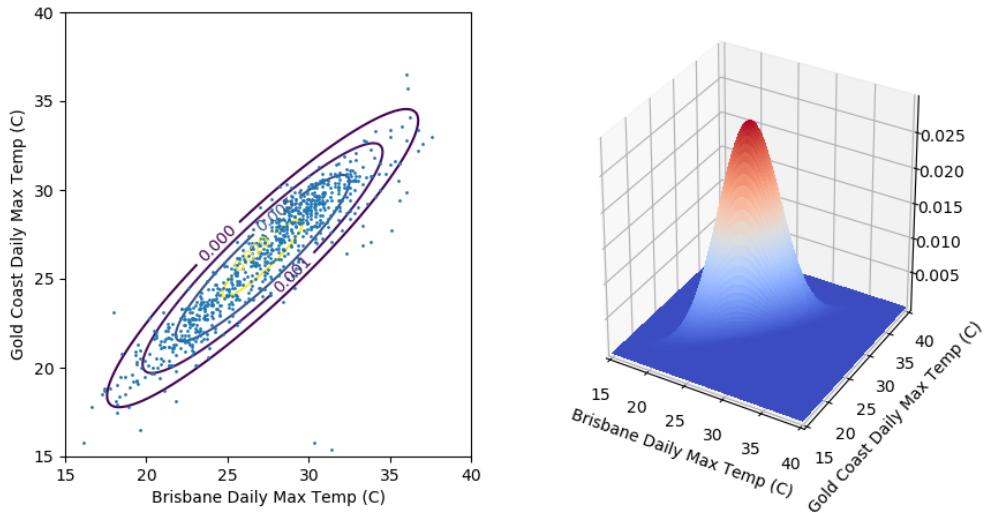


Figure 3.26: Contour lines and a surface plot for a bivariate normal distribution with randomly generated points on the contour plot.

```

16
17 ax = fig[:add_subplot](122, projection = "3d")
18 plot_surface(support, support, Z, rstride=1, cstride=1, lw=0,
19               antialiased=false, cmap="coolwarm",);

```

- In line 3 we include another Julia file defining `meanVect` and `covMat`.
- In line 4 we create an `MvNormal` distribution object representing the bivariate distribution.
- In line 7 we use `rand()` (with a method provided via the `Distributions` package to generate random points.
- The rest of the code deals with plotting. Notice the call to `contour()` on line 15, with specified levels.

Chapter 4

Processing and Summarizing Data - DRAFT

In this chapter we introduce methods, techniques and Julia examples for processing and summarizing data. In statics nomenclature, this is known as *descriptive statistics*. In the data-science nomenclature, such activities take the names of *analytics* and *dash-boarding*, while the process of manipulating and pre-processing data is sometimes called *data cleansing*, or *data cleaning*.

The statistical techniques and tools that we introduce include summary statistics and methods of visualization. We introduce several computational tools including the Julia DataFrames package, which allows for the storage of datasets that contain non-homogeneous data, with empty or missing entries, and the StatsBase package, which contains useful functions for summarizing data.

Statisticians and data-scientists can collect data in various ways, including *experimental studies*, *observational studies*, *longitudinal studies*, *survey sampling* and *data scraping*. Then in an effort to make inferences and conclusions from the data, one may look at the data via different *data configurations*. These configurations include:

Single sample: A case where all observations are considered to represent items from a homogeneous population. The configuration of the data takes the form: x_1, x_2, \dots, x_n .

Single sample over time (time series): The configuration of the data takes the form: $x_{t_1}, x_{t_2}, \dots, x_{t_n}$ with time points $t_1 < t_2 < \dots < t_n$.

Two samples: Similar to the single sample case, only now there are two populations (x 's and y 's). The configuration of the data takes the form: x_1, \dots, x_n and y_1, \dots, y_m .

Generalizations from two samples to k samples (each of potentially different sample size, n_1, \dots, n_k).

Observations in tuples: In this case, while the configuration of the data may seem similar to the two sample case, each observation is a tuple of points, (x, y) . Hence the configuration of data is

$(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$.

Generalizations from tuples to vectors of observations.

Other configurations including relationship data (graphs of connections), images, and many more possibilities.

In practice, data often initially comes in a raw format, and in such cases, one must first convert the format to one of the desired configurations above. In Section 4.1 below, we provide an example of this process.

This chapter is structured as follows: In Section 4.1 we see how to manipulate data frames in Julia. In Section 4.2 we deal with methods of summarizing data including basic elements of descriptive statistics. In Section 4.3 we present various ways of plotting data. Then Section 4.4 focuses on kernel density estimation. It is followed by Section 4.5 where we view ways of plotting cumulative probabilities including the empirical cumulative distribution function (ECDF). Finally, in Section 4.6 we see several ways of handling files in Julia.

4.1 Data Frames and Cleaning Data

In cases where data is homogeneous, arrays and matrices are used. However, more commonly, datasets are heterogeneous in nature, or contain incomplete or missing entries. In addition, datasets are often large, and commonly require “cleaning” before being stored. In such cases, arrays and matrices become inefficient storage mechanisms, and sometimes cannot be used at all.

The Julia *DataFrames* package introduces a data storage structure known as a `DataFrame`, which is aimed at overcoming these challenges. It can be used to store columns of different types, and also introduces the `missing` variable type which, as the name suggests, is used in place of missing entries.

The `missing` type has an important property, in that it “poisons” other types it interacts with. For example, if `x` represents a value, then `x + missing = missing`. This ‘poisoning’ effect ensures that missing values do not ‘infect’ and skew results when operations are performed on our data. For example, if `mean()` is used on a column with a missing value present, the result will evaluate as `missing`.

`DataFrames` are easy to work with. They can be created by using the `readtable()` function to import data directly from a `*.csv` or `*.txt` file, and columns and rows can be referenced by their position index, name (i.e symbol), or according to a set of user-defined rules. The `DataFrames` package also contains a variety of useful commands that can be used in conjunction with data frames, many of which we cover below.

Data Frames Step by Step

We now introduce `DataFrames` through the exploration and formatting of an example dataset. The data has five fields; `Name`, `Date`, `Time`, `Type` and `Price`. In addition, as is often the case with real datasets, there are missing values present in our data. Therefore, before analysis can start, some *data cleaning* must be performed.

Any variable in a dataset can be classified as either a *numerical variable*, or *categorical variable*. A numerical variable is a variable in which an ordered measurement is involved, such as height, weight, or IQ. A categorical variable on the other hand is any variable which communicates some information based on categories, or characteristics via group. Categorical variables can be further split into *nominal variable*, such as blood type, car model, or peoples names, and *ordinal variable*, in which some order is communicated, such as grades on a test, A to E, or a rating “Very high” to “Very low”. In our example, Price is a numerical variable, while Name is a nominal categorical variable. Since, in our example, Type can be thought of as a rating (A being best, and E being worst), Type would be an ordinal categorical variable, since it communicates some information about order.

We begin our example in Listing 4.1 below, where we load our data from the data file `purchaseData.csv`, and create our data frame. Note that in order to ensure each code block in this section runs as a standalone item, the `include()` function is used to initialize our data frame at the start of each Listing from 4.2 to 4.4.

Listing 4.1: Creating a DataFrame

```
1  using DataFrames, CSV
2
3  purchaseData = CSV.read("purchaseData.csv")
```

- In line 1 we load the `DataFrames` package, which allows us to use `DataFrame` type object.
- In line 3 we use the `readtable()` function to create a data frame object from our csv file. Note that by default, both the start and end of the data frame will be printed, however here we have suppressed the output via ; .

Following this, in Listing 4.2 below, we investigate the nature of our data.

Listing 4.2: Overview of a DataFrame

```
1  include("dataframeCreation.jl")
2  println(first(purchaseData, 6))
3  println(last(purchaseData, 6))
4  println(describe(purchaseData))
```

Row	me	Date	Time	Type	Price
1	MARYAN	14/09/2008	12:21 AM	E	8403
2	REBECCA	11/03/2008	8:56 AM	missing	6712
3	ASHELY	5/08/2008	9:12 PM	E	7700
4	KHADIJAH	2/09/2008	10:35 AM	A	missing
5	TANJA	1/12/2008	12:30 AM	B	19859
6	JUDIE	17/05/2008	12:39 AM	E	8033
Col #	Name	Eltypes	Missing	Values	
1	Name	Union{Missing, String}	13	MARYAN...RIVA	
2	Date	Union{Missing, String}	0	14/09/2008...30/12/2008	
3	Time	Union{Missing, String}	5	12:21 AM...5:48 AM	
4	Type	Union{Missing, String}	10	E...B	
5	Price	Union{Int64, Missing}	14	8403...15432	

- In line 1, the `include()` function is used so that the lines from Listing 4.1 are run first.
- In line 2, the `head()` function is used to display the first several rows of our data frame.
- In line 3 the `showcols()` function is used to display summary information about each column in the data frame. Specifically, each columns number, name (i.e. symbol), type, and the number of missing values in each column.
- Note there are many other useful functions which can be used with data frames. These include; `tail()`, which returns the last several rows of a data frame, `size()`, which returns the dimensions of the data frame, `names()`, which returns a vector of all column names as symbols, and `describe()`, which returns summary information about each column.
- Note that further documentation is available via `?DataFrame`.

It can be seen that there are quite a few missing values in our data, and we will return to this problem soon. However, first we cover how to reference values within a data frame. Values can be referenced by row and column index, and columns can also be referenced via their symbol, for example `:Price`. Symbols are a powerful concept used in metaprogramming, and we look at this further later in this section .

First however, we cover some basic examples of referencing data frames in Listing 4.3 below.

Listing 4.3: Referencing data in a DataFrame

```

1  include("dataframeCreation.jl")
2  println(purchaseData[13:17, [:Name]])
3  println(purchaseData.Name[13:17])
4  purchaseData[ismissing.(purchaseData.Time), :]
5  filter(row-> ismissing(row.Time), purchaseData)

```

```

5x1 DataFrame
| Row | Name      |
-----
| 1   | SAMMIE    |
| 2   | missing    |
| 3   | STACEY    |
| 4   | RASHIDA   |
| 5   | MELINA    |

5-element Array{Union{Missing, String},1}:
 "SAMMIE"
missing
"STACEY"
"RASHIDA"
"MELO"

5x5 DataFrame
| Row | Name     | Date       | Time      | Type | Price |
-----
| 1   | MING     | 10/11/2008 | missing    | B    | 6492  |
| 2   | JUSTI    | 19/07/2008 | missing    | C    | 16299 |
| 3   | ARDATH   | 13/12/2008 | missing    | D    | 26582 |
| 4   | KATTIE   | 12/10/2008 | missing    | D    | 19270 |
| 5   | HERMINE  | 17/09/2008 | missing    | C    | 27929 |

```

- Line 2 prints a `DataFrame` type object, containing values contained in the `Names` column for rows 13 to 17. Note the way the rows and column references are wrapped in square brackets here `[]`, and that the column was referenced via its symbol (i.e. `:Name`).
- Line 3 by comparison, returns a 1-dimensional `Array`, containing values from the `Names` column for rows 13 to 17. Note the different style of referencing used to that on line 2. In this case two separate brackets `[]` were used, the column was referenced first, and the rows second. Note also that the column could have been referenced via symbol (i.e. `:Name`) as in line 2.
- In line 5, the `ismissing()` function is used to return only rows which have missing values in the `Time` column. In this example, `ismissing.(purchaseData[:Date])` is used to check all rows of the `Time` for missing values, and only rows that satisfy this condition are returned. Note the use of the colon, `:`, which ensures that all columns are returned.
- An important point to note, although not shown here, both vertical and horizontal concatenation work the same way as matrices.

Now that we are somewhat more familiar with how referencing works, we return to the problem of missing values. As we have seen, missing values are recorded using the `missing` type, and it is this ability to have a placeholder in the case of missing entries that makes `DataFrames` so useful. As discussed at the start of this section, the `missing` type “poisons” other types, and this property ensures that missing values do not “infect” and skew results when operations are performed on a dataset. However, care must still be taken when applying logical operations.

For example, say we wish to find the mean of all prices of our dataset, excluding `missing` values. In Listing 4.4 we perform this operation, and highlight the importance of using the correct functions and logic when parsing data.

Listing 4.4: Dealing with missing type entries

```

1  using Statistics
2
3  include("dataframeCreation.jl")
4  println(mean(purchaseData.Price))
5  println(mean(dropmissing(purchaseData).Price))
6  println(mean(skipmissing(purchaseData.Price)))

```

```

missing
16616.925925925927
16384.483870967742

```

- In line 2 we attempt to calculate the mean of the `Price` column, however since `missing` values are present, the result is “poisoned” and `missing` is returned.
- In line 3, the `dropmissing()` function is used on `purchaseData`. It creates a copy of the data frame, that excludes all rows with `missing` entries . Then the `Price` column of this data frame is then used as an input to the `mean()` function. Note that importantly, in this example, all rows with `missing` entries are excluded, not just the rows for which `Price` is `missing`. Hence, some rows which had price records were not included in our `mean()` calculation, and in the case of our example, results in an incorrect calculation.

- In line 4, the `Price` column of `purchaseData` is returned first, and then the `skipmissing()` function applied to this data, and finally the `mean()` calculated. In this example, only rows with missing values in the `Price` column were excluded from our `mean()` calculation, hence this is the correct implementation of our intended logic.

In reality, depending on how many values are missing, it may not always be practical to simply exclude all of them. Instead, one way of dealing with missing values is to use *imputation*, which involves substituting missing values with entries. Care must be taken when imputing, as this approach can lead to bias in the data. Various methods of imputation exist, however we will use several simple techniques to replace missing entries in our dataset. In the listing below, we carry out this data cleaning by performing the following steps in order:

1. Delete all rows which have missing in both the `Type` and `Price` columns
2. Replace missing in the `Name` column with the string `notRecorded`.
3. Replace missing's in the `Time` column with the string `12:00 PM`
4. Replace remaining missing's in the `Type` column with a uniformly and randomly selected type from $\{A, B, C, D, E\}$.
5. Based on existing values in the `Price` column, calculate the mean price for each individual group of $\{A, B, C, D, E\}$, and replace all missing's with these value.
6. Save our newly imputed data via the `writetable()` function.

Listing 4.5: Cleaning and imputing data

```

1  using Random, Statistics
2  Random.seed!(0)
3  include("dataframeCreation.jl")
4
5  filter!(purchaseData) do row
6      !(ismissing(row.Type) && ismissing(row.Price))
7  end
8
9  replace!(purchaseData.Name, missing=>"notRecorded")
10 replace!(purchaseData.Time, missing=>"12:00 PM")
11
12 types = unique(skipmissing(purchaseData.Type))
13 replace!(x -> ismissing(x) ? rand(types) : x, purchaseData.Type)
14
15 for g in groupby(purchaseData, :Type)
16     prices = skipmissing(g.Price)
17     isempty(prices) || replace!(g.Price, missing=>round(Int, mean(prices)))
18 end
19
20 CSV.write("purchaseDataImputed.csv", purchaseData)
21 describe(purchaseData)

```

194x5 DataFrames.DataFrame	Col #	Name	Eltype	Missing	Values	

1	me	Union{Missing, String}	0	MARYAN...RIVA
2	Date	Union{Missing, String}	0	14/09/2008...30/12/2008
3	Time	Union{Missing, String}	0	12:21 AM...5:48 AM
4	Type	Union{Missing, String}	0	E...B
5	Price	Union{Int64, Missing}	0	8403...15432

4.2 Summarizing Data

Now that we have introduced data frames and methods of data processing, and organized our data into a usable configuration, we apply classical statical methods for data summaries.

Single Sample

Given a set of observations (data), x_1, \dots, x_n , we can compute a variety of descriptive statistics. The *sample mean*, denoted \bar{x} is given by,

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n}.$$

The *sample variance* is,

$$s^2 = \frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n-1} = \frac{\sum_{i=1}^n x_i^2 - n\bar{x}^2}{n-1}.$$

The *sample standard deviation* s is related to the sample variance via, $s := \sqrt{s}$. Also of interest, is the *standard error*, $\text{error} = s/\sqrt{n}$. Other descriptive statistics involve *order statistics* based on sorting the data, with the *sorted sample* sometimes denoted by,

$$x_{(1)} \leq x_{(2)} \leq \dots \leq x_{(n)}.$$

This allows us to define a variety of statistics such as the *minimum* ($x_{(1)}$), *maximum* ($x_{(2)}$), and the *median*, which in the case of n being odd is $x_{((n+1)/2)}$ (and in case of n being even an adjusted value). Related statistics are the α -*quantile*, for $\alpha \in [0, 1]$ which is effectively, $x_{(\tilde{\alpha}n)}$ (where $\tilde{\alpha}n$ is a rounding of αn to the nearest integer, or an interpolation). Note that for $\alpha = 0.25$ or $\alpha = 0.75$, these values are known as the *first quartile* and *third quartile* respectively. Finally the *inter quartile range (IQR)* is the difference between these two quartiles.

In Julia, these functions are implemented in either Julia base, the standard libraries (specifically the `Statistics` package, or the `StatsBase` package. In Listing 4.6 below, we use various Julia defined functions to calculate summary statistics on a single sample data set, that contains percentage grades from a test (0-100).

Listing 4.6: Summary statistics

```
1  using Statistics, StatsBase
```

```

2  data = parse.(Int, readlines("grades.csv"))
4
5  xbar = mean(data)
6  svar = var(data)
7  sdev = std(data)
8  minval = minimum(data)
9  maxval = maximum(data)
10 med = median(data)
11 per95 = percentile(data, 95)
12 q95 = quantile(data, 0.95)
13 intquartrng = iqr(data)
14
15 println("Sample Mean: $xbar")
16 println("Sample Variance: $svar")
17 println("Sample Standard Deviation: $sdev")
18 println("Minimum: $minval")
19 println("Maximum: $maxval")
20 println("Median: $med")
21 println("95th percentile: $per95")
22 println("0.95 quartile: $q95")
23 println("Interquartile range: $intquartrng")
24
25 summarystats(data)

```

```

Sample Mean: 52.08
Sample Variance: 950.3266666666668
Sample Standard Deviation: 30.827368792465354
Minimum: 5
Maximum: 96
Median: 61.0
95th percentile: 22.0
0.95 quartile: 75.0
Interquartile range: 53.0
Summary Stats:
Mean: 52.080000
Minimum: 5.000000
1st Quartile: 22.000000
Median: 61.000000
3rd Quartile: 75.000000
Maximum: 96.000000

```

- In line 3 we use the `readcsv()` function to load our data. Note that the argument type of the column is specified as `Int`, and that the data starts in the first row (i.e. there is no header). Also note the trailing `[:, 1]`, which is used so that the data is stored as an array, rather than as a data frame.
- In lines 5 to 13, we use the following functions to calculate the statistics described previously; `mean()`, `var()`, `std()`, `minimum()`, `maximum()`, `median()`, `percentile()`, `quantile()`, and `iqr()`.
- In lines 15 to 23 we print the results of the functions used.
- In line 25 the `summarystats()` function from the `StatsBase` package is used, which returns the mean, minimum, maximum, median along with the 1st and 3rd quartiles.

Observations in Tuples

When data is configured in the form of tuples, specifically of two observations, $(x_1, y_1), \dots, (x_n, y_n)$, we often consider the *sample covariance*, which is given by,

$$\text{cov}_{x,y} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{n - 1}.$$

Another useful statistic is the *sample correlation*, which is given by,

$$\hat{\rho} := \frac{\text{cov}_{x,y}}{s_x s_y},$$

where s_x and s_y are the sample standard deviations of the samples x_1, \dots, x_n and y_1, \dots, y_n respectively.

In the case of pairs of observations, these numbers can be represented in a 2×2 *sample covariance matrix* as follows,

$$\hat{\Sigma} = \begin{bmatrix} s_x^2 & \text{cov}_{x,y} \\ \text{cov}_{x,y} & s_y^2 \end{bmatrix},$$

where s_x^2 and s_y^2 are the sample variances.

In Listing 4.7, we import a weather observation dataset, containing pairs of temperature observations (see Section 3.7). We then estimate the elements of the covariance matrix, and then store the results in file `mvParams.jl`. Note this file is used as input in Listing 3.36 at the end of Chapter 3.

Listing 4.7: Estimating elements of a covariance matrix

```

1  using DataFrames, CSV, Statistics
2
3  data = CSV.read("temperatures.csv")
4  brisT = data[:, 4]
5  gcT = data[:, 5]
6
7  sigB = std(brisT)
8  sigG = std(gcT)
9  covBG = cov(brisT, gcT)
10
11 meanVect = [mean(brisT) , mean(gcT) ]
12 covMat = [sigB^2 covBG
13           covBG sigG^2]
14
15 outfile = open("mvParams.jl", "w")
16 write(outfile, "meanVect = $meanVect \ncovMat = $covMat")
17 close(outfile)
18 print(read("mvParams.jl", String))

```

```

meanVect = [27.1554, 26.1638]
covMat = [16.1254 13.047; 13.047 12.3673]

```

- In lines 3 to 5, we import our temperature data. The temperature data for Brisbane and the Gold Coast is stored as the arrays `brisT` and `gcT` respectively.
- In lines 7 to 8 the standard deviations of our temperature observations are calculated, and in line 9 the `cov()` function is used to calculate the covariance.
- In line 11, the means of our temperatures are calculated, and stored as an array, `meanVect`.
- In line 13 to 15, the covariance matrix is calculated and assigned to the variable `covMat`.
- In lines 16 to 19 we save `meanVect` and `covMat` to the new Julia file, `mvParams.jl`. Note that this file is used as input for our calculations in Listing 3.36.
- First, in line 16 the `open()` function is used (with the argument `w`) to create the file `mvParams.jl` in write mode. Note that `open()` creates an input-output stream, `outfile`, which can then be written to.
- Then in line 17 `write` function is used to write to the input-output stream `outfile`.
- In line 20, the input-output stream `outfile` is closed.
- In line 19, the content of the file `mvParams.jl` is printed via the `read` and `print` functions.

Vectors of Observations

We now consider data that consists of n vectors of observations. That is where the i 'th data point represents a tuple of values, (x_{i1}, \dots, x_{ip}) . Hence, the data can be represented by a $n \times p$ matrix.

It is often of interest to calculate, the $p \times p$ *sample covariance matrix*, $\hat{\Sigma}$, (a generalization of the 2×2 case previously covered). This sample covariance matrix serves as an estimator for a covariance matrix in equation 3.12, , of a random vector, (X_1, \dots, X_p) as described in Section 3.7. We can represent the matrix of observations via,

$$\mathbf{x} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1p} \\ x_{21} & \dots & \dots & x_{2p} \\ \vdots & & & \vdots \\ x_{n1} & x_{n2} & \dots & x_{np} \end{bmatrix}.$$

We can then define a matrix $\bar{\mathbf{x}}$ via,

$$\bar{\mathbf{x}} = \begin{bmatrix} \bar{x}_1 & \bar{x}_2 & \dots & \bar{x}_p \\ \bar{x}_1 & \dots & \dots & \bar{x}_p \\ \vdots & & & \vdots \\ \bar{x}_1 & \bar{x}_2 & \dots & \bar{x}_p \end{bmatrix},$$

where \bar{x}_i is the sample mean for the i 'th column. Then, combining the two, can can express the sample covariance matrix via the following calculation,

$$\hat{\Sigma} = \frac{1}{n-1}(\mathbf{x} - \bar{\mathbf{x}})'(\mathbf{x} - \bar{\mathbf{x}}).$$

In Julia, this calculation can be performed by through the use of the `cov()` function on the matrix `x`. We now illustrate this in Listing 4.8 below.

Listing 4.8: Sample covariance

```

1  using Statistics
2
3  X = [0.9 2.1 1.2;
4      1.1 1.9 2.5;
5      1.7 1.9 3.4;
6      0.8 2.3 2.3;
7      1.3 1.6 9.4;
8      0.7 2.7 1.3;
9      0.9 2.1 4.4]
10
11 n,p = size(X)
12
13 xbar = [mean(X[:,i]) for i in 1:p]'
14 ourCov = (X .- xbar)'*(X .- xbar)/(n-1)
15
16 println(ourCov)
17 println(cov(X))

```

```
[0.119524 -0.087381 0.44;
 -0.087381 0.121429 -0.715;
  0.44 -0.715 8.03333]
```

```
[0.119524 -0.087381 0.44;
 -0.087381 0.121429 -0.715;
  0.44 -0.715 8.03333]
```

- In line 11 the `size()` function returns a tuple with entries assigned to the individual variables, `n` and `p`.
- In line 13, note the use of the transpose operator, `'`. Initially the comprehension of means is treated as an array and hence a column vector. The transposition makes it a row vector.
- In line 14, note the use of `.-` for representing an element wise subtraction between a matrix and a vector. That is, for every row of `X` we subtract the elements of `xbar`. Hence our code doesn't explicitly create the matrix \bar{x} .
- Line 17 calculates `cov(X)` using the built-function. As observed from the output, this matches our manual calculation.

4.3 Plotting Data

The ability to plot data allows us to visually draw conclusions, and interpret and communicate patterns from data. In this section, we provide several plotting examples. The `PyPlot` package contains many standard plotting functions, some of which we have already seen. Here we will look at several as yet unseen functions, and then provide an example of our own custom plot.

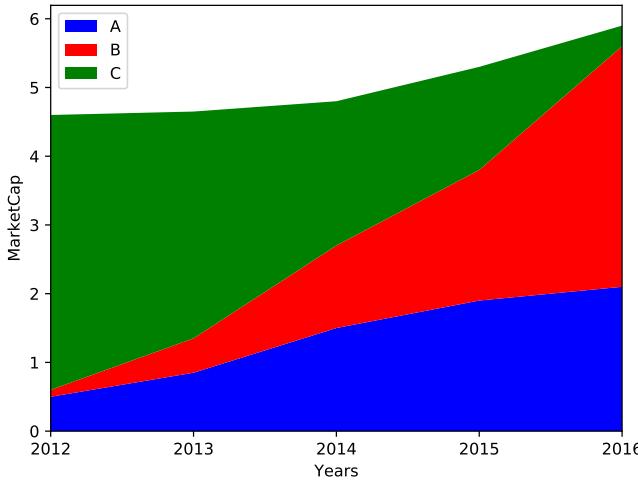


Figure 4.1: A stack plot showing the change in market cap of several companies over time.

In the following we present a *stack plot*. For this example, we consider a sample data set for three companies; A, B and C (see `companyData.csv`). In Listing 4.9 below, we create a stack plot which shows how the Market Cap of each company has changed over time.

Listing 4.9: A stack plot

```

1  using DataFrames, CSV, PyPlot
2
3  df = CSV.read("companyData.csv")
4
5  types = unique(df[:Type])
6  @assert length(unique(df[df.Type .== t, :Year] for t in types)) == 1
7  years = df[df.Type .== "A", :Year]
8  sizeA = df[df.Type .== "A", :MarketCap]
9  sizeB = df[df.Type .== "B", :MarketCap]
10 sizeC = df[df.Type .== "C", :MarketCap]
11
12 stackplot(years, sizeA, sizeB, sizeC, colors=["b", "r", "g"], labels = types)
13 legend(loc="upper left")
14 xlim(minimum(years),maximum(years))
15 xticks(years)
16 xlabel("Years")
17 ylabel("MarketCap")

```

- In line 3 we import our data as a data frame, via the `readtable()` function.
- In line 5, we obtain an array of each type of company, from the `Type` column.
- In line 7 we obtain an array of years over which we will plot our data. Note in this example, the period is identical for each company.
- In lines 8 to 10 we obtain arrays of the market cap of each company, for companies A, B and C.

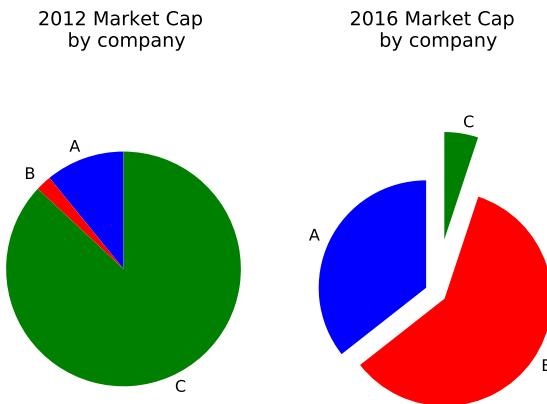


Figure 4.2: Pie charts.

- In line 12 the `stackplot()` function is used to create our plot. Note that the first argument is used to plot the x-axis co-ordinates, and the following arguments are equally sized arrays, containing the market cap data for each firm. Note also that for each tick on the x-axis, the market cap values are stacked, an array of formatting colors is specified, and the values in `types` used as the labels of the x-axis.
- We observe that over time, company C has shrunk in size, while companies A and B have expanded to fill the hole in the market. Note that overall the market itself has only marginally increased.

We now look at another commonly seen graph, the pie graph. In Listing 4.10 below, we construct two pie graphs showing the relative MarketCap for the years 2012 and 2016 for our companies A, B and C.

Listing 4.10: A pie chart

```

1  using DataFrames, CSV, PyPlot
2
3  df = CSV.read("companyData.csv")
4
5  types = unique(df.Type)
6
7  year2012 = df[df.Year .== 2012, :MarketCap]
8  year2016 = df[df.Year .== 2016, :MarketCap]
9
10 subplot(121)
11 pie(year2012, colors=["b", "r", "g"], labels = types, startangle=90)
12 axis("equal")
13 title("2012 Market Cap \n by company")
14
15 subplot(122)
16 pie(year2016, colors=["b", "r", "g"], labels = types, explode=[0.1, 0.1, 0.5],
17      startangle=90)
18 axis("equal")
19 title("2016 Market Cap \n by company")

```

- In lines 1 to 8 we load our data and prepare the arrays, `year2012` and `year2016` using similar logic to that in Listing 4.9 previously covered.
- In lines 10 to 19, we plot our piecharts. In particular, in lines 11 and 16 the `pie` function is used to create a pie plot of the Market cap data for the years 2012 and 2016. Note the use of the optional arguments, `colors`, to format the color of the plot, `explode`, which has the effect of separating the sectors as shown on the plot on the right, and the `startangle` argument, which is used to rotate the plot so that the plot starts from a vertical position, rather than horizontally.

We now look at an example of a custom plot . In Listing 4.11 below, we use the same company data as in the examples above, and plot the stock price against the dividend of each company for the years recorded. In addition, the relative market cap of each company is also indicated by the size of each data point plotted. This example provides an illustration of how one may visualize multiple metrics on the same plot.

Listing 4.11: A custom scatterplot

```

1  using DataFrames, CSV, PyPlot
2
3  df = CSV.read("companyData.csv")
4
5  for g in groupby(df, :Type)
6      xVals = g.Dividend
7      yVals = g.StockPrice
8      mktCap = g.MarketCap
9      maxYear = maximum(g.Year)
10
11      scatter(xVals, yVals, mktCap*100, alpha=0.5)
12      plot(xVals, yVals, label = "Company $(g.Type[1])")
13      legend(loc="upper left")
14      annotate("$maxYear", xy = (last(xVals), last(yVals)))
15  end
16  xlabel("Dividend (%)")
17  ylabel("StockPrice (\$)")
18  xlim(0,10)
19  ylim(0,10)

```

- In line 5 we create the array `types`, which contains each company name.
- In Lines 7-17 we cycle through each company stored in the array `types` and, for each company, performs the following.
- In lines 8 and 9, stores the dividend and stock prices as arrays `xVals` and `yVals` respectively.
- In line 10, stores the market cap of each company in the array `mktCap`.
- In line 11 the last yearly observation (i.e. year) for each company is assigned to `maxYear`.
- Then in line 13, a scatterplot of the dividends and stock prices is created. Importantly, each point is scaled according to the relative size of the market cap of each company.
- In line 14, the data points of each company are connected.

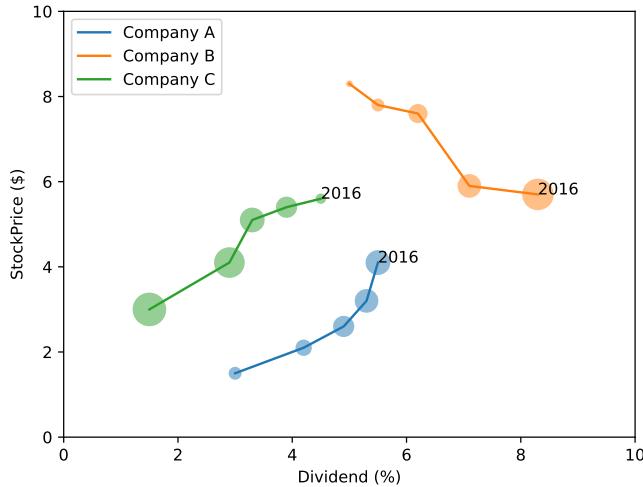


Figure 4.3: A custom plot showing how each company's stock price, dividend, and market cap changes over time.

- In line 15 the legend of the plot is set.
- In line 16. the last year for each companies data series is plotted.

4.4 Kernel Density Estimation

We now consider *kernel density estimation (KDE)*. This is a way of fitting a probability density function to a data-set. Given a set of observations, x_1, \dots, x_n , the KDE is the function,

$$\hat{f}(x) = \frac{1}{n} \sum_{i=1}^n \frac{1}{h} K\left(\frac{x - x_i}{h}\right), \quad (4.1)$$

where $K(\cdot)$ is some specified *kernel function* and $h > 0$ is the so called *bandwidth* parameter. The kernel function is a function that satisfies the properties of a PDF. A typical example is the Gaussian kernel.

$$K(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}.$$

The bandwidth parameter then applies a scaling by h . Closer examination of (4.1) indicates that the obtained density estimate, $\hat{f}(\cdot)$ is simply a superposition of scaled kernels, centred at each of the observations. Clearly if h is too large, the KDE tends to look just like $K(\cdot)$. Alternatively if h is very small, then the KDE will look like a bunch of spikes, with a spike at each data-point, x_i . This indicates that much of the emphasis of working with KDE is choosing a sensible bandwidth, h .

A (default) classic rule is called *Silverman's rule*. It is based on the sample standard deviation of the sample, s :

$$h = \left(\frac{4}{3}\right)^{1/6} s n^{-1/5} \approx 1.06 s n^{-1/5}.$$

There is some theory justifying this h in certain cases (and other theory indicating why sometimes other rules are better), however we won't get into this here. In Julia's KDE available via the `KernelDensity`, the default bandwidth uses this rule. Nevertheless you may alter the bandwidth if desired.

We now look at an example in Listing 4.12 below, in which we first create a *mixture model* of synthetic data, and then randomly sample data from this model. We then perform KDE on our sampled data, and plot a comparison between the estimated PDF, a histogram of our sampled data, and the analytic PDF of the underlying mixture model. It can be observed that the PDF approximated using KDE is closely aligned with the underlying PDF of our model.

Listing 4.12: Kernel density estimation

```

1  using Random, Distributions, KernelDensity, PyPlot
2  Random.seed!(0)
3
4  mu1, sigma1 = 10, 5
5  mu2, sigma2 = 40, 12
6
7  z1 = Normal(mu1,sigma1)
8  z2 = Normal(mu2,sigma2)
9
10 p = 0.3
11
12 function mixRv()
13     (rand() <= p) ? rand(z1) : rand(z2)
14 end
15
16 function actualPDF(x)
17     p*pdf(z1,x) + (1-p)*pdf(z2,x)
18 end
19
20 numSamples = 100
21 data = [mixRv() for _ in 1:numSamples]
22
23 xGrid = -20:0.1:80
24 pdfActual = actualPDF.(xGrid)
25 kdeDist = kde(data)
26 pdfKDE = pdf(kdeDist,xGrid)
27
28 plt[:hist](data,20, histtype = "step", normed=true, label="Sample data")
29 plot(xGrid,pdfActual,"-b",label="Underlying PDF")
30 plot(xGrid,pdfKDE, "-r",label="KDE PDF")
31 xlim(-20,80)
32 ylim(0,0.035)
33 xlabel(L"X")
34 legend(loc="upper right")

```

- In line 1 we load the required packages `Distributions`, `KernelDesnity`, and `PyPlot`.
- In line 2 we set the seed of the random number generator, which we require in this example for traceability.
- In lines 4 and 5 we specify the means and standard deviations for the two underlying normal distributions which our mixture model is composed of. The normal distributions are created

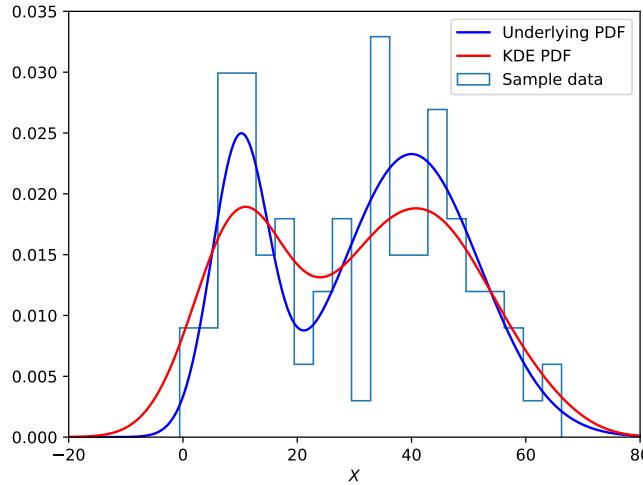


Figure 4.4: Comparing the population PDF to a histogram and a kernel density estimate from a sample.

in lines 7 and 8.

- In lines 12 to 14 we create the function `mixRv()`, which returns a random variable from our mixture model. It works as follows, first a uniform random number over $U[0, 1)$ is generated, and if this value is less than or equal to p (specified in line 10), then our data point is randomly generated from the distribution z_1 , else it is generated from z_2 .
- In lines 16 to 18 the function `actualPDF(x)` is created. This function returns the analytic solution of the underlying PDF of our mixture model, by using the `pdf()` function to evaluate the actual pdf, given at the specified point x .
- In lines 20 to 21 the `mixRV()` function is used to generate 100 random samples. The data is assigned to the array `data`.
- In lines 23 to 24 the actual PDF of the mixture model is calculated over the domain `xGrid`.
- In line 25 the function `kde()` is used to generate a KDE type object `kdeDist`, based on the data in `data`.
- In line 26, the kernel density estimated PDF is calculated over the domain `xGrid`.
- Lines 28 to 30 then plot a histogram of the sample data, the true underlying PDF of the mixture model, and the KDE generated pdf from our sample data. Note that KDE PDF is fairly close to the actual underlying PDF.

4.5 Plotting Cumulative Probabilities

Plotting cumulative probabilities as opposed to probability masses or densities is often very useful. We now look at a few cases.

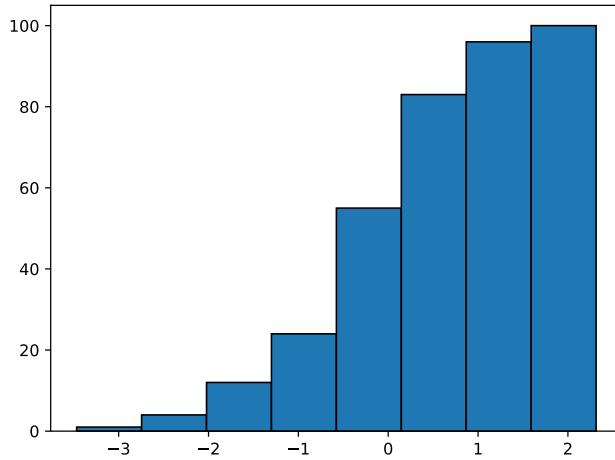


Figure 4.5: Cumulative histogram plot.

Cumulative Histogram

As we have previously seen, histograms are useful tools when it comes to seeing how sample data is distributed. In Listing 4.13 below, we plot a *cumulative histogram* for a small sample data set.

Listing 4.13: A cumulative histogram plot

```

1  using Random, Distributions, PyPlot
2  Random.seed!(0)
3
4  data = randn(10^2)
5  plt[:hist](data, 8, ec="black", cumulative=true);

```

- This plot is similar to the many histograms we have previously seen, however note the use of the argument `cumulative=true`, which results in a cumulative histogram plot.

Empirical Distribution Function

While KDE is a useful way to estimate the PDF of the unknown underlying distribution given some sample data, the *empirical distribution function* (ECDF) may be viewed as an estimate of the underlying CDF. The ECDF is a stepped function, which, given n data points, increases by $1/n$ at each point. Mathematically, given the sample, x_1, \dots, x_n the ECDF of this is given by,

$$\hat{F}_n(t) = \frac{1}{n} \sum_{i=1}^n \mathbf{1}\{x_i \leq t\} \quad \text{where } \mathbf{1}\{\cdot\} \text{ is the indicator function.}$$

Constructing an ECDF is possible in Julia through the `ecdf()` function contained in the `StatsBase`, and we now provide an example in Listing 4.14. Consider that we have some data

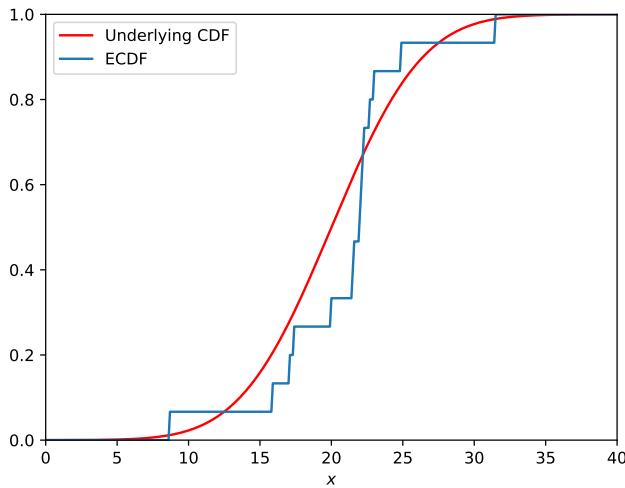


Figure 4.6: The ECDF from a sample compared to the population CDF.

points from an unknown underlying distribution. We create an ECDF object from that data, then plot a comparison between this, and the underlying CDF. A comparison of the two is shown in Figure 4.6.

Listing 4.14: Empirical cumulative distribution function

```

1  using Random, Distributions, StatsBase, PyPlot
2  Random.seed!(0)
3
4  underlyingDist = Normal(20, 5)
5  data = rand(underlyingDist, 15)
6
7  empiricalCDF = ecdf(data)
8
9  xGrid = 0:0.1:40
10 plot(xGrid,cdf(underlyingDist,xGrid),"-r",label="Underlying CDF")
11 plot(xGrid,empiricalCDF(xGrid),label="ECDF")
12 xlim(0,40)
13 ylim(0,1)
14 xlabel(L"x")
15 legend(loc="upper left");

```

- In line 1 we load our required packages.
- In lines 4 and 5, we define our underlying distribution, and sample from it 15 observations.
- In line 7, we use the `ecdf()` function on our sample data. Note that this generates an empirical distribution function, which can be used to evaluate the ecdf given an input value. We assign this new function as `empiricalDF`.
- In line 10 the actual underlying PDF from which our sample data was generated is plotted over the domain `xGrid`.
- In line 11, the `empiricalDF` function defined above is compared to the ECDF over the domain `xGrid`.

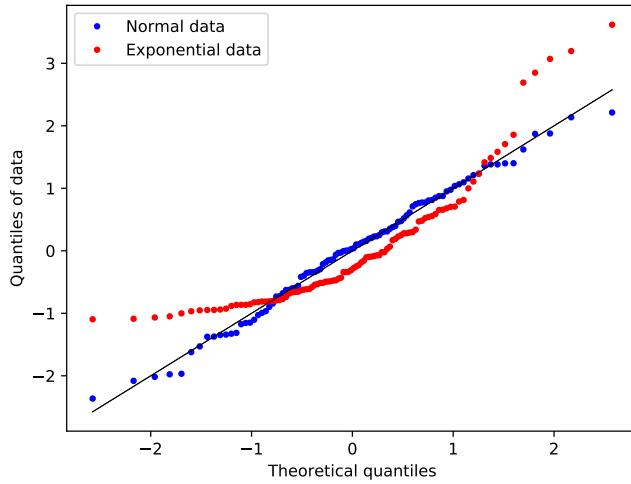


Figure 4.7: Comparing two normal probability plots. One from a normal population and one from an exponential distribution.

Normal Probability Plot

In Listing 4.16 below, we create a normal probability plot based on two sample data sets, the first coming from a normal distribution, and the second from an exponential distribution.

Listing 4.15: Normal probability plot

```

1  using Random, PyPlot, Distributions, StatsBase
2  Random.seed!(0)
3  function normalProbabilityPlot(data, C, L)
4      mu = mean(data)
5      sig = std(data)
6      n = length(data)
7      p = [(i-0.5)/n for i in 1:n]
8      x = quantile(Normal(),p)
9      y = sort([(i-mu)/sig for i in data])
10     plot(x, y, C, label=L)
11     xRange = maximum(x) - minimum(x)
12     plot( [minimum(x), maximum(x)],
13           [minimum(x), maximum(x)], "k", lw=0.5)
14     xlabel("Theoretical quantiles")
15     ylabel("Quantiles of data")
16     legend(loc="upper left")
17 end
18
19 normalData = randn(100)
20 expData = rand(Exponential(),100)
21
22 normalProbabilityPlot(normalData,".b","Normal data")
23 normalProbabilityPlot(expData,".r","Exponential data")

```

- It can be observed that the exponential distribution is not linear in nature, but rather the tails of the plot are skewed.

4.6 Working with Files

In this last section, we look at exporting and saving data. We first provide an example in Listing 4.16, where we create a function which searches a text document for a given keyword, and then saves every line of text containing this keyword to a new text file, along with the associated line number.

Listing 4.16: Filtering an input file

```

1  function lineSearch(inputFilename, outputFilename, keyword)
2      infile = open(inputFilename, "r")
3      outfile = open(outputFilename, "w")
4
5      for (index, line) in enumerate(split(read(infile, String), "\n"))
6          if occursin(keyword, line)
7              println(outfile, "$index: $line")
8          end
9      end
10     close(infile)
11     close(outfile)
12 end
13
14 lineSearch("earth.txt", "waterLines.txt", "water")

```

```

17: 71% of Earth's surface is covered with water, mostly by oceans. The
19: have many lakes, rivers and other sources of water that contribute to the

```

- Line 1 is where we begin to define our function `lineSearch`. As arguments, it takes an input file name, an output file name, and our search keyword.
- Line 2 uses the `open` function with the optional argument `"r"` to open our specified input file in read mode. It creates an `IOStream` type object, which we can use as arguments to other functions. We assign this as `infile`.
- Line 3 uses the `open` function with the optional argument `"w"` to create and open a file with our specified output name. This file is created on disk, ready to have information written to it. This `IOStream` object is assigned as `outfile`.
- Lines 5-9 contain a `for` loop, which is used to search through our `infile` for our specified keyword.
- Line 5 uses the `eachline()` function, which creates an iterable object, and is used in conjunction with the `enumerate` function, which is an iterator. When combined with the `for` loop as shown here, the `eachline()` function cycles through each line of our input file, and the `enumerate` function returns a tuple of values, which contains both the index and the value of that index (ie. the line number, and the corresponding text of that line, which are both referenced by `index` and `text` respectively).
- Line 6 uses the `occursin` function to check if the given line (`line`) contains our given keyword (`keyword`). If it does, then we proceed to line 7, else we continue the loop for the next line of our input file.

- Line 7 uses the `println` function to write both the index of the line found which contains our keyword, and the line text to the output file, `outfile`. Note that `println` is used, so that each line of text appears on a newline in our text file.
- Lines 10 and 11 close both our input file and output file.
- Line 14 is an example of our `lineSearch` function in action. Here we give the input file as "`earth.txt`", specify the output file as "`waterLines.txt`", and our searchable keyword as "`water`". Note that this function is case sensitive.

For our next example, we create a function which searches a directory for all filenames which contain a given search string. It then saves a list of these files to a file `fileList`. Note that this function does not behave recursively and only searches the directory given.

Listing 4.17: Searching files in a directory

```

1  function directorySearch(directory, searchString)
2      outfile  = open("fileList.txt", "w")
3      fileList = filter(x->occursin(searchString, x), readdir(directory))
4
5      for file in fileList
6          println(outfile, file)
7      end
8      close(outfile)
9  end
10
11 directorySearch(pwd(), ".jl")

```

- Line 1 is where we begin to define our function `directorySearch`. As arguments, it takes a directory to search through, and a `searchString`.
- Line 2 uses the `open` function with the argument "`w`" to create our output file `fileList.txt`, which we will write to.
- In line 3 we create a string array of all filenames in our specified `directory` that contain our `searchString`. This string array is assigned as `fileList`. The `readdir` function is used to list all files in the specified `directory`, and the `filter` function is used, along with the `occursin` function to check over each element if the string contains `searchString`.
- Lines 5-7 loop through each element in `fileList` and print them to our output file `outfile`.
- Line 8 closes the `IOStream` of our `outfile`.
- Line 11 provides an example of the use of our `directorySearch` function, where we use it to obtain a shortlist of all files whose extensions contain ".jl" within our current working directory (i.e. `pwd()`).

Chapter 5

Statistical Inference Ideas - DRAFT

This chapter introduces statistical inference ideas, with the goal of establishing a theoretical footing of key concepts that follow in later chapters. The action of *statistical inference* involves using mathematical techniques to make conclusions about unknown *population* parameters based on collected data. The field of statistical inference employs a variety of stochastic models to analyze and put forward efficient and plausible methods for carrying out such analyses.

In broad generality, analysis and methods of statistical inference can be categorized as either *frequentist* (also known as classical) or *Bayesian*. The former is based on the assumption that population parameters of some underlying distribution, or probability law, exist and are fixed, but are yet unknown. The process of statistical inference then deals with making conclusions about these parameters based on sampled data. In the latter Bayesian case, it is only assumed that there is a *prior distribution* of the parameters. In this case, the key process deals with analyzing a *posterior distribution* (of the parameters) - an outcome of the inference process. In this book we focus almost solely on the classical frequentist approach with the exception of Section 5.7 where we explore Bayesian statistics briefly.

In general, a statistical inference process involves *data*, a *model*, and *analysis*. The data is assumed to be comprised of random samples from the model. The goal of the analysis is then to make informed statements about population parameters of the model based on the data. Such statements typically take one of the following forms:

Point estimation - Determination of a single value (or vector of values) representing a best estimate of the parameter/parameters. In this case, the notion of “best” can be defined in different ways.

Confidence intervals - Determination of a range of values where the parameter lies. Under the model and the statistical process used, it is guaranteed that the parameter lies within this range with a pre-specified probability.

Hypothesis tests - The process of determining if the parameter lies in a given region, in the complement of that region, or fails to take on a specific value. Such tests often represent a scientific hypothesis in a very natural way.

Most of the point estimation, confidence intervals and hypothesis tests that we introduce and

carry out in this book are very elementary. Chapter 6 is devoted to covering elementary confidence intervals in detail, and Chapter 7 is devoted to covering elementary hypothesis tests in detail. We now begin to explore key ideas of statistical inference.

This chapter is structured as follows: In Section 5.1 we present the concept of a random sample together with the distribution of statistics such as the sample mean and sample variance. In Section 5.2 we focus on random samples of normal random variables. In this common case, certain statistics have well known distributions that play a central role in statistics. In Section 5.3 we explore the central limit theorem, providing justification for the ubiquity of the normal distribution. In Section 5.4 we explore basics of point estimation. In Section 5.5 we explore the concept of a confidence interval. In Section 5.6 we explore concepts of hypothesis testing. Finally, in Section 5.7 we explore basics of Bayesian statistics.

5.1 A Random Sample

When carrying out (frequentist) statistical inference, we assume there is some underlying distribution $F(x; \theta)$ from which we are sampling, where θ is the scalar or vector valued unknown parameter we wish to know. Importantly, we assume that each observation is statistically independent and identically distributed as the rest. That is, from a probabilistic perspective, the observations are taken as *independent and identically distributed (i.i.d.)* random variables. In mathematical statistics language, this is called a *random sample*. We denote the random variables of the observations by X_1, \dots, X_n and their respective values by x_1, \dots, x_n .

Typically, we compute *statistics* from the random sample. For example, two common standard statistics include the *sample mean* and *sample variance* introduced in Section 4.2 in the context of data summary. However, we can model these *statistics* as random variables.

$$\bar{X} = \frac{1}{n} \sum_{i=1}^n X_i, \quad \text{and} \quad S^2 = \frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})^2. \quad (5.1)$$

In general, the phrase *statistic* implies a quantity calculated based on the sample. When working with data, the sample mean and sample variance are nothing but numbers computed from our sample observations. However, in the statistical inference paradigm, we associate random variables to these values, since they are themselves functions of the random sample. We look at properties of such statistics, and see how they play a role in estimating the unknown underlying distribution parameter θ .

Note that for S^2 , the denominator is $n - 1$ (as opposed to n as one might expect). This makes S^2 an *unbiased estimator*. We discuss this property further in Section 5.4.

To illustrate the fact that \bar{X} and S^2 are random variables, assume we have sampled data from an exponential distribution with $\lambda = 4.5^{-1}$ (a mean of 4.5 and a variance of 20.25). If we collect $n = 10$ observations, then the sample mean and sample variance are non-trivial random variables. In Listing 5.1 below, we investigate their distribution through Monte Carlo simulation.

Listing 5.1: Distributions of the sample mean and sample variance

```
1   using Random, Distributions, PyPlot
```

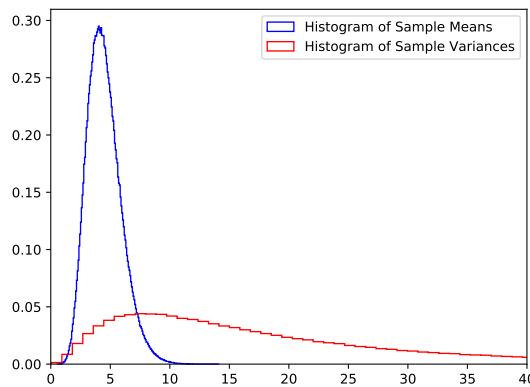


Figure 5.1: Histograms of the sample mean and sample variance of an exponential distribution.

```

2 Random.seed!(0)
3
4 lambda = 1/4.5
5 expDist = Exponential(1/lambda)
6 n, N = 10, 10^6
7
8 means = Array{Float64}(undef, N)
9 variances = Array{Float64}(undef, N)
10
11 for i in 1:N
12     data = rand(expDist, n)
13     means[i] = mean(data)
14     variances[i] = var(data)
15 end
16
17 plt[:hist](means, 200, color="b", label = "Histogram of Sample Means",
18             histtype = "step", density = true)
19 plt[:hist](variances, 600, color="r", label = "Histogram of Sample Variances",
20             histtype = "step", density = true)
21 xlim(0, 40)
22 legend(loc="upper right")
23 println("Actual mean: ", mean(expDist),
24         "\nMean of sample means: ", mean(means))
25 println("Actual variance: ", var(expDist),
26         "\nMean of sample variances: ", mean(variances))

```

```

Actual mean: 4.5
Mean of sample means: 4.500154606762812
Actual variance: 20.25
Mean of sample variances: 20.237117004185237

```

- In lines 11-15 we simulate N sample means and sample variances from our exponential distribution, by taking n observations each time. The sample means and variances are stored in the arrays `means` and `variances` respectively.
- In lines 17-20, we generate histograms of the sample means and sample variances, using 200

and 600 bins respectively. It can be seen that the sample means and variances we generated are indeed random variables in themselves.

- In lines 23-26, we calculate the mean of both arrays means and variances. It can be seen that the means of our simulated data are good approximations to the mean and variance parameters of the underlying exponential distribution. That is for an exponential distribution with rate λ the mean is λ^{-1} and the variance is λ^{-2} .

5.2 Sampling from a Normal Population

It is often assumed that the distribution $F(x; \theta)$ is a Normal distribution, and hence $\theta = (\mu, \sigma^2)$. This assumption is called the *normality assumption*, and is sometimes justified due to the central limit theorem, which we cover in Section 5.3 below. Under the normality assumption, the distribution of the random variables \bar{X} and S^2 are well known:

$$\begin{aligned}\bar{X} &\sim \text{Normal}(\mu, \sigma^2/n), \\ (n-1)S^2/\sigma^2 &\sim \chi_{n-1}^2, \\ T := \frac{\bar{X} - \mu}{S/\sqrt{n}} &\sim t_{n-1}.\end{aligned}\tag{5.2}$$

Here ‘ \sim ’ denotes ‘*distributed as*’, and implies that the statistics on the left hand side of the ‘ \sim ’ symbols are distributed according to the distributions on the right hand side. The notation χ_{n-1}^2 and t_{n-1} denotes a *chi-squared* and *student T-distribution* respectively, each with $n-1$ degrees of freedom. The chi-squared distribution is a Gamma distribution (see Section 3.6) with parameters $\lambda = 1/2$ and $\alpha = n/2$. The student T-distribution is introduced in the subsection below.

Importantly, these distributional properties of the statistics from a normal sample theoretically support the statistical procedures that are presented in Chapters 6 and 7.

We now look at an example in Listing 5.2 below, where we sample data from a normal distribution and compute of the statistics, \bar{X} and S^2 . We then show that the distribution of sample means, sample variances and t-statistics (T) indeed follow the distributions given by equation (5.2).

Listing 5.2: Friends of the normal distribution

```

1  using Distributions, PyPlot
2
3  mu, sigma = 10, 4
4  n, N = 10, 10^6
5
6  sMeans = Array{Float64}(undef, N)
7  sVars = Array{Float64}(undef, N)
8  tStats = Array{Float64}(undef, N)
9
10 for i in 1:N
11     data      = rand(Normal(mu,sigma),n)
12     sampleMean = mean(data)
13     sampleVars = var(data)
14     sMeans[i]  = sampleMean
15     sVars[i]   = sampleVars

```

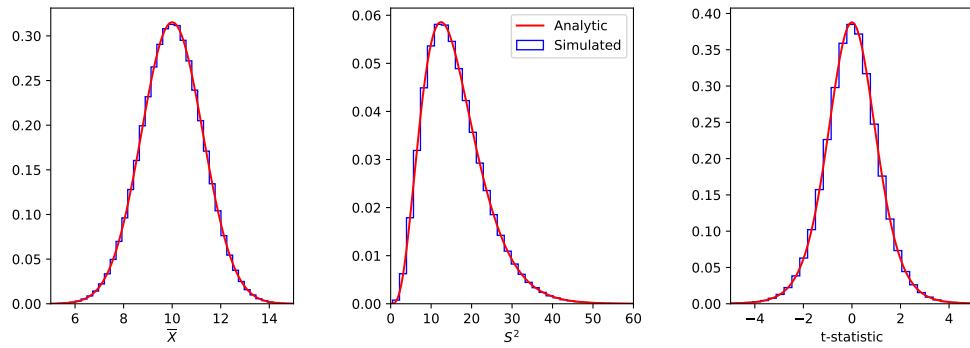


Figure 5.2: Histograms of the simulated sample means, sample variances, and t-statistics, against their analytic counterparts.

```

16      tStats[i] = (sampleMean - mu) / (sqrt(sampleVars/n))
17  end
18
19 xRangeMean = 5:0.1:15
20 xRangeVar = 0:0.1:60
21 xRangeTStat = -5:0.1:5
22
23 figure(figsize=(12.4,4))
24 subplots_adjust(wspace=0.4)
25
26 subplot(131)
27 plt[:hist](sMeans,50, histtype="step", color="b", normed = true)
28 plot(xRangeMean, pdf(Normal(mu,sigma/sqrt(n)),xRangeMean), "-r")
29 xlim(5,15)
30 xlabel(L"\overline{X}")
31
32 subplot(132)
33 plt[:hist](sVars,50, histtype="step", color="b", label="Simulated",
34             normed=true);
35 plot(xRangeVar, (n-1)/sigma^2*pdf(Chisq(n-1), xRangeVar * (n-1)/sigma^2),
36       "-r", label="Analytic")
37 xlim(5,15)
38 xlim(0,60)
39 xlabel(L"S^2")
40 legend(loc="upper right")
41
42 subplot(133)
43 plt[:hist](tStats,80, histtype="step", color="b", normed=true);
44 plot(xRangeTStat, pdf(TDist(n-1), xRangeTStat), "-r",)
45 xlim(5,15)
46 xlim(-5,5)
47 xlabel("t-statistic")

```

- In line 3, we specify the parameters of the underlying distribution, from which we sample our data.
- In line 4 we specify the number of samples taken in each group n, and the total number of groups N.

- In lines 6–9 we initialize three arrays, which will be used to store our sample means, variances, and t-statistics.
- In lines 10–17, we conduct our numerical simulation, by taking n sample observations from our underlying normal distribution, and calculating the sample mean, sample variance, and t-statistic. This process is repeated N times, and the values stored in the arrays `sMeans`, `sVars`, and `tStats` respectively.
- In lines 26–47 the histograms of our sample means, sample variances, and t-statistics are plotted alongside the analytic PDF's given by equation (5.2).
- In line 28, we plot the analytic PDF of the sample mean. This is a simple process of plotting the PDF of a normal distribution with a mean of 10, and a standard deviation of $4/\sqrt{2}$.
- In line 35 and 36 we plot the analytic PDF of a scaled Chi-squared distribution through the use of the `pdf()` and `Chisq()` functions. Note that the values on the x-axis and the density are both normalized by $(n-1)/\sigma^2$ to reflect the fact we are interested in the PDF of a scaled Chi-squared distribution.
- In line 44 the analytic PDF of the t-statistic (T), which is described by a T-distribution, is plotted via the use of the `TDist()` function. We cover the T-distribution at the end of this section.

Independence of the Sample Mean and Sample Variance

Consider a random sample, X_1, \dots, X_n . In general, one would not expect the sample mean, \bar{X} and the sample variance S^2 to be independent random variables - since both of these statistics rely on the same underlying values. For example, consider a random sample where $n = 2$, and let each X_i be Bernoulli distributed, with parameter p . The joint distribution of \bar{X} and S^2 is then simple to compute:

If both X_i 's are 0 (happens w.p. $(1-p)^2$): $\bar{X} = 0$ and $S^2 = 0$.

If both X_i 's are 1 (happens w.p. p^2): $\bar{X} = 1$ and $S^2 = 0$.

If one of the X_i 's is 0, and the other is 1 (happens w.p. $2p(1-p)$): $\bar{X} = 1/2$ and $S^2 = 1 - 2(\frac{1}{2})^2 = 1/2$.

Hence, as shown in Figure 5.3 the joint PMF of \bar{X} and S^2 is,

$$\mathbb{P}(\bar{X} = \bar{x}, S^2 = s^2) = \begin{cases} (1-p)^2, & \text{for } \bar{x} = 0 \text{ and } s^2 = 0, \\ 2p(1-p), & \text{for } \bar{x} = 1/2 \text{ and } s^2 = 1/2, \\ p^2, & \text{for } \bar{x} = 1 \text{ and } s^2 = 0. \end{cases} \quad (5.3)$$

Further, the (marginal) PMF of \bar{X} is,

$$\mathbb{P}_{\bar{X}}(0) = (1-p)^2, \quad \mathbb{P}_{\bar{X}}\left(\frac{1}{2}\right) = 2p(1-p), \quad \mathbb{P}_{\bar{X}}(1) = p^2.$$

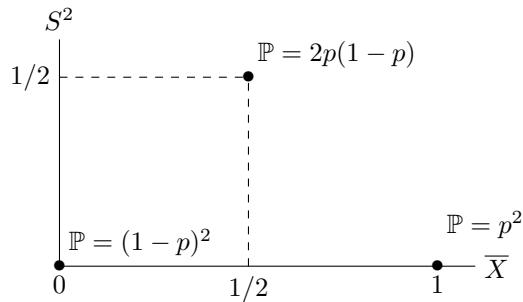


Figure 5.3: PMF of sample mean and sample variance.

And the (marginal) PMF of S^2 is,

$$\mathbb{P}_{S^2}(0) = (1-p)^2 + p^2, \quad \mathbb{P}_{S^2}\left(\frac{1}{2}\right) = 2p(1-p),$$

We now see that \bar{X} and S^2 are not independent because the joint distribution,

$$\tilde{\mathbb{P}}(i, j) = \mathbb{P}_{\bar{X}}(i) \mathbb{P}_{S^2}(j), \quad i, j \in \{0, \frac{1}{2}, 1\},$$

constructed by the product of the marginal distributions does not equal the joint distribution in (5.3).

The example above demonstrates dependence between \bar{X} and S^2 . This is in many ways unsurprising. However importantly, in the special case where the samples, X_1, \dots, X_n are from a normal distribution, independence between \bar{X} and S^2 does hold. In fact, this property characterizes the normal distribution - that is, this property only holds for the normal distribution, see [Luk42].

We now explore this concept further in Listing 5.3 below. In it we compare a standard normal distribution to what we call a standard uniform distribution - a uniform distribution on $[-\sqrt{3}, \sqrt{3}]$ which exhibits zero mean and unit variance. For both of these distributions, we consider a random sample of size n , and from this obtain the pair (\bar{X}, S^2) . We then plot points of these pairs against points of pairs where \bar{X} and S^2 are each obtained from two separate sample groups. In Figure 5.4 it can be seen that for the Normal distribution, regardless of whether the pair (\bar{X}, S^2) is calculated from the same sample group, or from two different sample groups, the points appear to behave similarly (this is because they have the same joint distribution). However, for the standard uniform distribution, it can be observed that the points behave in a completely different manner.

Listing 5.3: Are the sample mean and variance independent?

```

1  using Distributions,PyPlot
2
3  function statPair(dist,n)
4      sample = rand(dist,n)
5      [mean(sample),var(sample)]
6  end
7
8  stdUni = Uniform(-sqrt(3),sqrt(3))

```

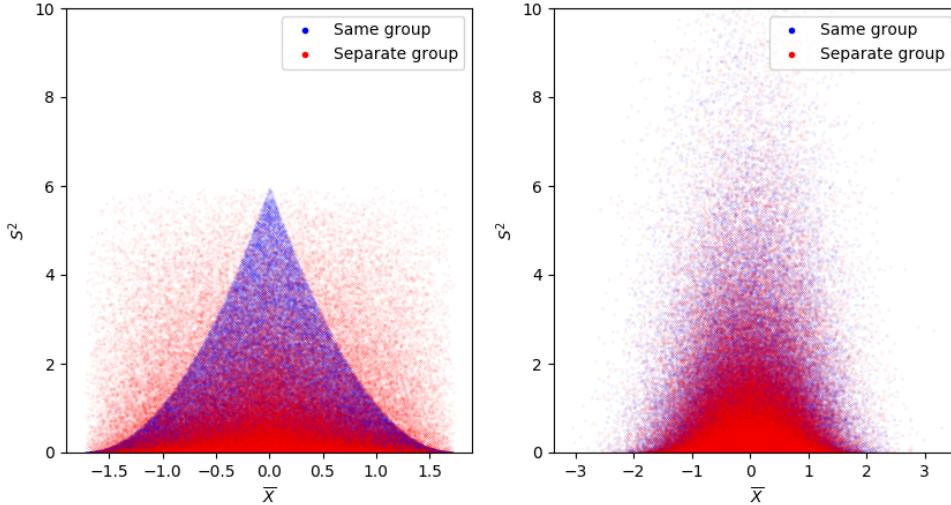


Figure 5.4: Pairs of \bar{X} and S^2 for standard uniform (left) and standard normal (right). Blue points are for statistics calculated from the same sample, and red for statistics were calculated from separate samples.

```

9 n, N = 2, 10^5
10
11 dataUni      = [statPair(stdUni,n) for _ in 1:N]
12 dataUniInd   = [[mean(rand(stdUni,n)),var(rand(stdUni,n))] for _ in 1:N]
13 dataNorm     = [statPair(Normal(),n) for _ in 1:N]
14 dataNormInd  = [[mean(rand(Normal(),n)),var(rand(Normal(),n))] for _ in 1:N]
15
16 figure("test", figsize=(10,5))
17 subplot(121)
18 plot(first.(dataUni),last.(dataUni),"b",ms="0.1",label="Same group")
19 plot(first.(dataUniInd),last.(dataUniInd),"r",ms="0.1", label="Separate group")
20 xlabel(L"\overline{X}")
21 ylabel(L"S^2")
22 legend(markerscale=60,loc="upper right")
23 ylim(0,10)
24
25 subplot(122)
26 plot(first.(dataNorm),last.(dataNorm),"b",ms="0.1",label="Same group")
27 plot(first.(dataNormInd),last.(dataNormInd),"r",ms="0.1", label="Separate group")
28 xlabel(L"\overline{X}")
29 ylabel(L"S^2")
30 legend(markerscale=60,loc="upper right")
31 ylim(0,10)

```

- In lines 3-6 the function `statPair()` is defined. It takes a distribution and integer `n` as input, and generates a random sample of size `n`, and then returns the sample mean and sample variance of this random sample as an array.
- In line 8 we define the standard uniform distribution, which has a mean of zero and a standard deviation of 1.

- In line 9 we specify that n observations will be made for each sample, and that the total number of sample groups is N .
- In line 11, the function `statPair()` is used along with a comprehension to calculate N pairs of sample means and variances from N sample groups. Note that the observations are all sampled from the standard uniform distribution `stdUni`, and that the output is an array of arrays.
- In line 12 a similar approach to line 11 is used. However, in this case, rather than calculating the sample mean and variance from the same sample group each time, they are calculated from two separate sample groups N times. As before, the data is sampled from the standard uniform distribution `stdUni`.
- Lines 13 and 14 are identical to lines 11-12, however in this case observations are sampled from a standard normal distribution `Normal()`.
- The remaining lines are used to plot the output. Note the use of the `first()` and `last()` functions along with the element wise operator `.` on lines 18, 19, 26 and 27. These are used together to extract the first and last elements, the sample means and sample variances respectively.
- From Figure 5.4 we can see that in the case of the standard uniform distribution, if the sample mean and variance are calculated from the same sample group, then all pairs of \bar{X} and S^2 fall within a specific bounded region. The envelope of this blue region can be clearly observed, and represents the region of all possible combinations of \bar{X} and S^2 when calculated based on the same sample data. On the other hand, if \bar{X} and S^2 are calculated from two separate samples, then we observe a scattering of data, shown by the points in red. This difference in behavior shows that in this case \bar{X} and S^2 are not independent, but rather the outcome of one imposes some restriction on the outcome of the other.
- By comparison, in the case of the standard normal distribution, regardless of how the pair (\bar{X}, S^2) are calculated, (from the same sample group, or from two different groups) the same scattering of points is observed, supporting the fact that \bar{X} and S^2 are independent.

More on the T-Distribution

Having explored the fact that \bar{X} and S^2 are independent in the case of a Normal sample, we now elaborate on the *Student T-distribution* and focus on the distribution of the *T-statistic*, that appeared earlier in (5.2). The random variable is given by:

$$T = \frac{\bar{X} - \mu}{S/\sqrt{n}}.$$

Denoting the mean and variance of the individual observations by μ and σ^2 respectively, we can represent the T-statistic as,

$$T = \frac{\sqrt{n}(\bar{X} - \mu)/\sigma}{\sqrt{nS^2/\sigma^2 n}} = \frac{Z}{\sqrt{\chi^2/n}}.$$

Here the numerator Z is a standard normal random variable and in the denominator, $\chi^2 = nS^2/\sigma^2$ is chi-squared distributed with n degrees of freedom (as claimed in (5.2)). Further, the numerator

and denominator random variables are independent because they are based on the sample mean and sample variance.

Observing that the random variable T can be represented as a ratio of a standard normal random variable and an independent normalized root of a chi-squared random variable, we obtain some insight. Notice that $\mathbb{E}[\chi^2] = n$ and $\text{Var}(\chi^2) = 2n$. The variance of χ^2/n is $2/n$, and hence one may expect that as $n \rightarrow \infty$, the random variable χ^2/n gets more and more concentrated around 1, with the same holding for $\sqrt{\chi^2/n}$. Hence for large n one may expect the distribution of T to be similar to the distribution of Z , which is indeed the case. This plays a role in the confidence intervals and hypothesis tests in the chapters that follow.

Analytically, the distribution of T can be shown to have a density function,

$$f(x) = \frac{\Gamma\left(\frac{n+1}{2}\right)}{\sqrt{n\pi}\Gamma\left(\frac{n}{2}\right)} \left(1 + \frac{x^2}{n}\right)^{-\frac{n+1}{2}}$$

where the parameter n is the number of observations, also called the *degrees of freedom*. Note the presence of the gamma function, $\Gamma(\cdot)$, which is defined in Section 3.6. The T-distribution is a symmetric distribution with a “bell-curved” shape similar to that of the normal distribution, with “heavier tails” for non-large n .

In practice, when carrying out elementary statistical inference using the T-distribution (as presented in the following chapters), the most commonly used attribute is the quantile, covered in Section 3.3. Typically denoted by $t_{n,\alpha}$, these quantiles are often tabulated in standard statistical tables with α being a metric describing the quantile in question.

In Listing 5.4 below the PDFs of several T-distributions are plotted, illustrating that as the degrees of freedom increase, the PDF converges to the standard normal PDF.

Listing 5.4: Student’s T-distribution

```

1  using Distributions, PyPlot
2
3  xGrid = -5:0.1:5
4  plot(xGrid, pdf(Normal(), xGrid), "k", label="Normal Distribution")
5  plot(xGrid, pdf(TDist(1), xGrid), "b.", label="DOF = 1")
6  plot(xGrid, pdf(TDist(3), xGrid), "r.", label="DOF = 3")
7  plot(xGrid, pdf(TDist(100), xGrid), "g.", label="DOF = 100")
8  xlim(-4,4)
9  ylim(0,0.5)
10 xlabel("X")
11 ylabel("Probability")
12 legend(loc="upper right")

```

- In line 4 we plot the PDF of the standard normal distribution.
- In lines 5 to 7, we plot several PDF’s of the T-distribution, with different degrees of freedom (1, 3 and 100). It can be observed that as the degrees of freedom increase, the T-distribution converges to the standard normal distribution.

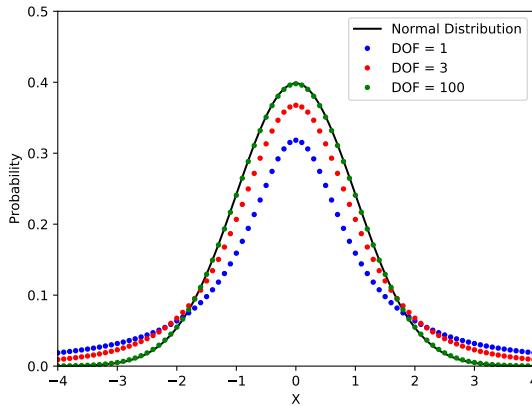


Figure 5.5: As the number of observations increases, the t-Distribution approaches that of the normal distribution.

Two Samples and the F-Distribution

Many statistical procedures involve looking at the ratio of sample variances, or similar quantities, for two or more samples. For example, if X_1, \dots, X_n is one sample and Y_1, \dots, Y_m is another sample, and both samples are distributed normally with the same parameters, one can look at the ratio of the two sample variances:

$$F_{\text{statistic}} = \frac{S_X^2}{S_Y^2}.$$

It turns out that such a statistic is distributed according to what is called the *F-distribution*, with density given by,

$$f(x) = K(n, m) \frac{x^{n/2-1}}{(m+nx)^{(n+m)/2}} \quad \text{with} \quad K(n, m) = \frac{\Gamma\left(\frac{n+m}{2}\right) n^{n/2} m^{m/2}}{\Gamma\left(\frac{n}{2}\right) \Gamma\left(\frac{m}{2}\right)}.$$

Here the parameters n and m are the *numerator degrees of freedom* and *denominator degrees of freedom* respectively.

In agreement with (5.2), an alternative view is that the random variable F is obtained by the ratio of two independent Chi-squared random variables, normalized by their degrees of freedom. This distribution plays a key role in the popular Analysis of Variance (ANOVA) procedures, further explored in Section 7.3.

We now briefly explore the F-distribution in Listing 5.5 below, by simulating two sample sets of data with n_1 and n_2 observations respectively from a normal distribution. The ratio of the sample variances from the two distributions is then compared to the PDF of an F-distribution, with parameters $n_1 - 1$ and $n_2 - 1$.

Listing 5.5: Ratio of variances and the F-distribution

```

1  using Distributions, PyPlot
2
3  n1, n2 = 10, 15

```

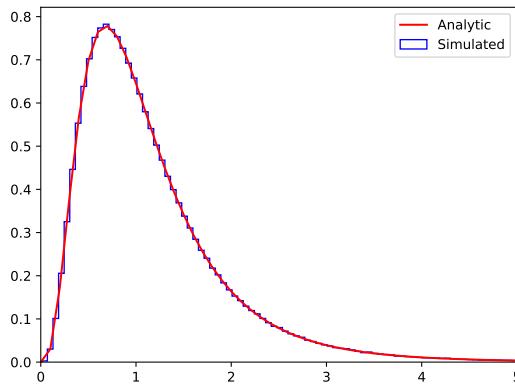


Figure 5.6: Histogram of the ratio of two sample variances against an F-distribution PDF.

```

4   N = 10^6
5   mu, sigma = 10, 4
6   normDist = Normal(mu,sigma)
7
8   fValues = Array{Float64} (undef, N)
9
10  for i in 1:N
11      data1 = rand(normDist,n1)
12      data2 = rand(normDist,n2)
13      fValues[i] = var(data1)/var(data2)
14  end
15
16  xRange = 0:0.1:5
17
18  plt[:hist](fValues,400,histtype="step",color="b",
19             label="Simulated",normed=true)
20  plot(xRange,pdf(FDist(n1-1, n2-1), xRange), "r",label="Analytic")
21  xlim(0,5)
22  legend(loc="upper right")

```

- In line 3-4 we define the total number of observations for our two sample groups, n_1 and n_2 , as well as the total number of F-statistics we will generate N .
- In lines 10 to 14 we simulate two separate sample groups, data1 and data2 , by randomly sampling from the same underlying normal distribution. A single F-statistic is then calculated from the ratio of the sample variances of the two groups.
- In lines 18 and 19 a histogram of the F-statistics is plotted using 400 bins total.
- In line 20 the function `FDist()` is used to plot the analytic PDF of an F-distribution with the parameters n_1 and n_2 . It can be seen that the histogram of the ratios of sample variances follow that of the F-distribution, with the parameters n_1-1 and n_2-1 .

5.3 The Central Limit Theorem

In the previous section we assumed sampling from a normal population, and this assumption gave rise to a variety of properties of statistics associated with the sampling. However, why would such an assumption hold? A key lies in one of the most exciting elementary results of probability and statistics: *the Central Limit Theorem* (CLT).

While the CLT has several versions and many generalizations, they all have one thing in common: summations of a large number of random quantities, each with finite variance, yields a sum that is approximately normally distributed. This is the main reason that the normal distribution is ubiquitous in nature and present throughout the universe.

We now develop this more formally. Consider an i.i.d. sequence X_1, X_2, \dots where all X_i are distributed according to some distribution $F(x_i ; \theta)$ with mean μ and finite variance σ^2 . Consider now the random variable,

$$Y_n := \sum_{i=1}^n X_i.$$

It is clear that $\mathbb{E}[Y] = n\mu$ and $\text{Var}(Y) = n\sigma^2$. Hence we may consider a random variable,

$$\tilde{Y}_n := \frac{Y_n - n\mu}{n\sigma^2},$$

with zero mean and unit variance. The CLT states that as $n \rightarrow \infty$, the distribution of \tilde{Y}_n converges to a standard normal distribution. That is, for every $x \in \mathbb{R}$,

$$\lim_{n \rightarrow \infty} \mathbb{P}(\tilde{Y}_n \leq x) = \int_{-\infty}^x \frac{1}{\sqrt{2\pi}} e^{-\frac{u^2}{2}} du.$$

Alternatively, this may be viewed as indicating that for non-small n ,

$$Y_n \underset{\text{approx}}{\sim} N(\mu, \sigma^2)$$

where N is a normal random variable with mean μ and variance σ^2 .

In addition, by dividing the numerator and denominator of \tilde{Y}_n by n , we see an immediate consequence of the CLT. That is, for non-small n , the sample mean of n observations denoted by \bar{X}_n satisfies,

$$\bar{X}_n \underset{\text{approx}}{\sim} N\left(\mu, \left(\frac{\sigma}{\sqrt{n}}\right)^2\right).$$

Hence the CLT states that sample means from i.i.d. samples with finite variances are asymptotically distributed according to a normal distribution as the sample size grows. This ubiquity of the normal distribution justifies the normality assumption employed when using many of the statistical procedures that we cover in Chapters 6 and 7.

To illustrate the CLT, consider the three different distributions below, noting that each has a mean and variance both equal to 1:

1. A Uniform distribution, on $[1 - \sqrt{3}, 1 + \sqrt{3}]$.

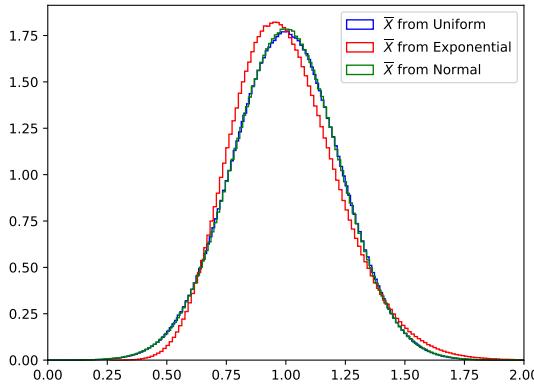


Figure 5.7: Histograms of sample means for different underlying distributions.

2. An exponential distribution with $\lambda = 1$.
3. A Normal distribution with both a mean and variance of 1.

In Listing 5.6 below, we illustrate the central limit theorem, by generating a histogram of N sample means for each of the three different distributions mentioned above. Although each of the underlying distributions are very different, (uniform, exponential and normal), the sampling distribution of the sample means all approach that of the normal distribution centered about 1.

Listing 5.6: The central limit theorem

```

1  using Distributions, PyPlot
2
3  n, N = 20, 10^7
4
5  dist1 = Uniform(1-sqrt(3),1+sqrt(3))
6  dist2 = Exponential(1)
7  dist3 = Normal(1,1)
8
9  data1 = [mean(rand(dist1,n)) for _ in 1:N]
10 data2 = [mean(rand(dist2,n)) for _ in 1:N]
11 data3 = [mean(rand(dist3,n)) for _ in 1:N]
12
13 plt[:hist](data1,200,color="b",label="Uniform",histtype="step",normed="True")
14 plt[:hist](data2,200,color="r",label="Exponential",histtype="step",normed="True")
15 plt[:hist](data3,200,color="g",label="Normal",histtype="step",normed="True")
16 xlim(0,2)
17 legend(loc="upper right")

```

- In lines 5-7 we define three different distribution type objects; a continuous uniform distribution over the domain $[1 - \sqrt{3}, 1 + \sqrt{3}]$, an exponential distribution with a mean of 1, and a normal distribution with mean and standard deviation both 1.
- In lines 9-11, we generate N sample means, (each consisting of n observations), for each distribution defined above.

- In lines 13-15 we plot three separate histograms based on the sample mean vectors previously generated. It can be observed that for large N, these histograms approach that of a normal distribution, and in addition, the mean of the data approaches the mean of the underlying distribution from which the samples were taken.

5.4 Point Estimation

A common task of statistical inference is to estimate a parameter, θ or a function of it, say $h(\theta)$, given a random sample, X_1, \dots, X_n . The process of designing an estimator, analyzing its performance, and carrying out the estimation is called *point estimation*.

Although we can never know the underlying parameter, θ , or $h(\theta)$ exactly, we can arrive at an estimate for it via an *estimator* $\hat{\theta} = f(X_1, \dots, X_n)$. Here the design of the estimator is embodied by $f(\cdot)$, a function that specifies how to construct the estimate from the sample.

An important question to ask is how close is $\hat{\theta}$ to the actual unknown quantity, θ or $h(\theta)$. In this section we first describe several ways of quantifying and categorizing this “closeness”, and then present two common methods for designing estimators; the *method of moments* and *maximum likelihood estimation* (MLE).

The design of (point) estimators is a central part of statistics, however in elementary statistics courses for science students, engineers, or social studies researchers, point estimation is often not explicitly mentioned. The reason for this is that for almost any common distribution, one can estimate the mean and variance via, \bar{X} and S^2 respectively, see (5.1). That is, in the case of $h(\cdot)$ being either the mean or the variance of the distribution, the estimator given by the sample mean or sample variance respectively is a natural candidate and performs exceptionally well. However, in other cases, choosing an estimation procedure is less straight forward.

Consider for example a case of a uniform distribution on the range $[0, \theta]$. Say we are interested in estimating θ based on a random sample, X_1, \dots, X_n . In this case here are a few alternative estimators:

$$\begin{aligned}\hat{\theta}_1 &= f_1(X_1, \dots, X_n) := \max\{X_i\}, \\ \hat{\theta}_2 &= f_2(X_1, \dots, X_n) := 2\bar{X}, \\ \hat{\theta}_3 &= f_3(X_1, \dots, X_n) := 2 \text{ median}(X_1, \dots, X_n), \\ \hat{\theta}_4 &= f_4(X_1, \dots, X_n) := \sqrt{12S^2}.\end{aligned}\tag{5.4}$$

Each of these makes some sense in its own right; $\hat{\theta}_1$ is based on the fact that θ is an upper bound of the observations, $\hat{\theta}_2$ and $\hat{\theta}_3$ utilize the fact that the sample mean and sample median are expected to fall on $\theta/2$, and finally $\hat{\theta}_4$ utilizes the fact that the variance of the distribution is given by $S^2 = \theta^2/12$. Given that there are various possible estimators, we require methodology for comparing them and perhaps developing others, with the aim of choosing a suitable one.

We now describe some methods for analyzing the performance of such estimators and others.

Describing the Performance and Behavior of Estimators

When analyzing the performance of an estimator $\hat{\theta}$, it is important to understand that it is a random variable. One common measure of its performance is the *Mean Squared Error* (MSE),

$$MSE_{\theta}(\hat{\theta}) := \mathbb{E}[(\hat{\theta} - \theta)^2] = \text{Var}(\hat{\theta}) + (\mathbb{E}[\hat{\theta}] - \theta)^2 := \text{variance} + \text{bias}^2. \quad (5.5)$$

The second equality arises naturally from adding and subtracting $\mathbb{E}[\hat{\theta}]$. In this representation, we see that the MSE can be decomposed into the variance of the estimator, and its bias squared. Low variance is clearly a desirable performance measure. The same applies to the *bias*, which is a measure of the expected difference between the estimator and the true parameter value. One question this raises is: are there cases where estimators are *unbiased* - that is, they have a bias of 0, or alternatively $\mathbb{E}[\hat{\theta}] = \theta$? The answer is yes. We show this now using the sample mean as a simple example.

Consider X_1, \dots, X_n distributed according to any distribution with a finite mean μ . In this case, say we are interested in estimating $\mu = h(\theta)$. It is easy to see that the sample mean \bar{X} is itself a random variable with mean μ , and is hence unbiased. Further, the variance of this estimator is σ^2/n , where σ^2 is the original variance of X_i . Since the estimator is unbiased, the MSE equals the variance, i.e. σ^2/n .

Now consider a case where the population mean μ is known, but the population variance, σ^2 , is unknown, and that we wish to estimate it. As a sensible estimator, consider:

$$\hat{\sigma}^2 := \frac{1}{n} \sum_{i=1}^n (X_i - \mu)^2. \quad (5.6)$$

Computing the mean of $\hat{\sigma}^2$ yields:

$$\mathbb{E}[\hat{\sigma}^2] = \frac{1}{n} \mathbb{E} \left[\sum_{i=1}^n (X_i - \mu)^2 \right] = \frac{1}{n} \sum_{i=1}^n \mathbb{E}[(X_i - \mu)^2] = \frac{1}{n} n \sigma^2 = \sigma^2.$$

Hence $\hat{\sigma}^2$ is an unbiased estimator for σ^2 . However, say we are now also interested in estimating the (population) standard deviation, σ . In this case it is natural to use the estimator,

$$\hat{\sigma} := \sqrt{\hat{\sigma}^2} = \sqrt{\frac{1}{n} \sum_{i=1}^n (X_i - \mu)^2}.$$

Interestingly, while this is a perfectly sensible estimator, it is not unbiased. We illustrate this via simulation in Listing 5.7 below. In it we consider a uniform distribution over $[0, 1]$, where the population mean, variance and standard deviation are 0.5 , $1/12$ and $\sqrt{1/12}$ respectively. We then estimate the bias of $\hat{\sigma}^2$ and $\hat{\sigma}$ via Monte Carlo simulation. The output shows that $\hat{\sigma}$ is not unbiased. However, as the numerical results illustrate, it is *asymptotically unbiased*. That is, the bias tends to 0 as the sample size n grows.

Listing 5.7: A biased estimator

```

1  using Random
2  Random.seed!(0)

```

```

3
4  function estVar(n)
5      sample = rand(n)
6      sum((sample .- 0.5).^2)/n
7  end
8
9  N = 10^7
10 for n in 5:5:30
11     biasVar = mean([estVar(n) for _ in 1:N]) - 1/12
12     biasStd = mean([sqrt(estVar(n)) for _ in 1:N]) - sqrt(1/12)
13     println("n = ", n, " Var bias: ", round(biasVar, digits=5),
14           "\t Std bias: ", round(biasStd, digits=5))
15 end

```

```

n = 5 Var bias: -1.0e-5 Std bias: -0.00639
n = 10 Var bias: -0.0      Std bias: -0.00303
n = 15 Var bias: -1.0e-5 Std bias: -0.00198
n = 20 Var bias: -0.0      Std bias: -0.00147
n = 25 Var bias: 1.0e-5   Std bias: -0.00117
n = 30 Var bias: 1.0e-5   Std bias: -0.00098

```

- In lines 4-7 the function `estVar()` is defined, which implements equation (5.6).
- In lines 10-15, we loop over sample sizes $n = 5, 10, 15, \dots, 30$, and for each we repeat N sampling experiments, for which we estimate the biases for $\hat{\sigma}^2$ and $\hat{\sigma}$ respectively. These values are stored in the variables `biasVar` and `biasStd`, and the values of these two variables are output for each case of sample size n .
- The results show that while the estimator for the variance is unbiased, the estimator for the standard deviation is only asymptotically unbiased.

Having explored an estimator for σ^2 with μ known (as well as a digression to the estimator of σ in the same case), what would a sensible estimator for σ^2 be in the more realistic case where μ is not known? A natural first suggestion would be to replace μ in (5.6) with \bar{X} to obtain,

$$\tilde{S}^2 := \frac{1}{n} \sum_{i=1}^n (X_i - \bar{X})^2.$$

However, with a few lines of expectation computations, one can verify that,

$$\mathbb{E}[\tilde{S}^2] = \frac{n-1}{n} \sigma^2.$$

Hence it is biased (albeit asymptotically unbiased). This is then the reason that the preferred estimator, S^2 is actually,

$$S^2 = \frac{n}{n-1} \tilde{S}^2.$$

as in (5.1). That estimator is unbiased.

There are other important qualitative properties of estimators that one may explore. One such property is *consistency*. Roughly, we say that an estimator is consistent if it converges to the

true value as the number of observations grows to infinity. More can be found in mathematical statistics references such as [DS11] and [CB01]. The remainder of this section presents two common methodologies for estimating parameters; method of moments and maximum likelihood estimation, and a comparison of these two methodologies is presented at the end.

Method of Moments

The *method of moments* is a methodological way to obtain parameter estimates for a distribution. The key idea is based on moment estimators for the k 'th moment, $\mathbb{E}[X_i^k]$,

$$\hat{m}_k = \frac{1}{n} \sum_{i=1}^n X_i^k. \quad (5.7)$$

As a simple example, consider a uniform distribution on $[0, \theta]$. An estimator for the first moment ($k = 1$) is then, $\hat{m}_1 = \bar{X}$. Now denoting by X a typical random variable from this sample, for such a distribution $\mathbb{E}[X^1] = \theta/2$. Hence it is sensible to equate the moment estimator with the first moment expression to arrive at the equation,

$$\frac{\theta}{2} = \hat{m}_1.$$

Notice that this equation involves the unknown parameter, θ and the moment estimator obtained from the data. Then trivially solving for θ yields the estimator,

$$\hat{\theta} = 2\hat{m}_1.$$

Notice that this is exactly $\hat{\theta}_2$ from (5.4).

In cases where there are multiple unknown parameters, say K , we use the first K moment estimates to formulate a system of K equations and K unknowns. This system of equations can be written as,

$$\mathbb{E}[X^k ; \theta_1, \dots, \theta_k] = \hat{m}_k, \quad k = 1, \dots, K. \quad (5.8)$$

For many textbook examples (such as the uniform distribution case described above), we are able to solve this system of equations analytically, yielding a solution,

$$\hat{\theta}_k = g_k(\hat{m}_1, \dots, \hat{m}_K), \quad k = 1, \dots, K. \quad (5.9)$$

Here the functions $g_k(\cdot)$ describe the solution of the system of equations. However, it is often not possible to obtain explicit expressions for $g_k(\cdot)$. In these cases typically numerical techniques are used to solve the corresponding system of equations.

As an example, consider the triangular distribution with density,

$$f(x) = \begin{cases} \frac{2}{(b-a)(c-a)} \frac{x-a}{b-x}, & x \in [a, c], \\ \frac{2}{(b-a)(b-c)} \frac{b-x}{x-a}, & x \in [c, b]. \end{cases}$$

This distribution has support $[a, b]$, and a maximum at c . Note that the Julia triangular distribution function uses this same parameterization: `TriangularDist(a, b, c)`.

Now straightforward (yet tedious) computation yields the first three moments, $\mathbb{E}[X^1]$, $\mathbb{E}[X^2]$, $\mathbb{E}[X^3]$ as well as the system of equations for the method of moments:

$$\begin{aligned}\hat{m}_1 &= \frac{1}{3}(a + b + c), \\ \hat{m}_2 &= \frac{1}{6}(a^2 + b^2 + c^2 + ab + ac + bc), \\ \hat{m}_3 &= \frac{1}{10}(a^3 + b^3 + c^3 + a^2b + a^2c + b^2a + b^2c + c^2a + c^2b + abc).\end{aligned}\quad (5.10)$$

Generally, this system of equations is not analytically solvable. Hence, the method of moments estimator is given by a numerical solution to (5.10). In Listing 5.8 below, given a series of observations, we numerically solve this system of equations through the use of the `NLsolve` package, and arrive at estimates for the values of a , b , and c .

Listing 5.8: Point estimation via method of moments using a numerical solver

```

1  using Random, Distributions, NLsolve
2  Random.seed!(0)
3
4  a, b, c = 3, 5, 4
5  dist = TriangularDist(a,b,c)
6  n = 2000
7  samples = rand(dist,n)
8
9  m_k(k,data) = 1/n*sum(data.^k)
10 mHats = [m_k(i,samples) for i in 1:3]
11
12 function equations(F, x)
13     F[1] = 1/3*( x[1] + x[2] + x[3] ) - mHats[1]
14     F[2] = 1/6*( x[1]^2 + x[2]^2 + x[3]^2 + x[1]*x[2] + x[1]*x[3] +
15                  x[2]*x[3] ) - mHats[2]
16     F[3] = 1/10*( x[1]^3 + x[2]^3 + x[3]^3 + x[1]^2*x[2] + x[1]^2*x[3] +
17                  x[2]^2*x[1] + x[2]^2*x[3] + x[3]^2*x[1] + x[3]^2*x[2] +
18                  x[1]*x[2]*x[3] ) - mHats[3]
19 end
20
21 nlOutput = nlsolve(equations, [ 0.1; 0.1; 0.1])
22 println("Found estimates for (a,b,c) = ", nlOutput.zero)
23 println(nlOutput)

```

```

Found estimates for (a,b,c) = [5.00303, 3.99919, 3.00271]
Results of Nonlinear Solver Algorithm
 * Algorithm: Trust-region with dogleg and autoscaling
 * Starting Point: [0.1, 0.1, 0.1]
 * Zero: [5.00303, 3.99919, 3.00271]
 * Inf-norm of residuals: 0.000000
 * Iterations: 14
 * Convergence: true
   * |x - x'| < 0.0e+00: false
   * |f(x)| < 1.0e-08: true
 * Function Calls (f): 15
 * Jacobian Calls (df/dx): 13

```

- In line 1 the `NLsolve` package is called. This package contains numerical methods for solving non-linear systems of equations.
- In lines 4-7, we specify the parameters of the triangular distribution and the distribution itself `dist`. We also specify the total number of samples `n`, and generate our sample set of observations `samples`.
- In line 9, the function `m_k()` is defined, which implements equation (5.7), and in line 10, this function is used to estimate the first three moments, given our observations `samples`.
- In line 12-19, we set up the system of simultaneous equations within the function `equations()`. This specific format is used as it is a requirement of the `nlsolve()` function which is used later. The `equations()` function takes two arrays as input, `F` and `x`. The elements of `F` represent the left hand side of the series of equations (which are later solved for zero), and the elements of `x` represent the corresponding constants of the equations. Note that in setting up the equations from (5.10), the moment estimators are moved to the right hand side, so that the zeros can be found.
- In line 21, the `nlsolve()` function from the `NLsolve` package is used to solve the zeros of the function `equations!()`, given a starting position of `[0.1; 0.1; 0.1]`. In this example, since the Jacobian was not specified, it is computed by finite difference.
- In line 22, the zeros of our function is printed as output through the use of `.zero`, which is used to return just the zero field of the `nlsolve()` output.
- In line 23, the complete output from the function `nlsolve()` is printed as output.
- The output shows that the numerically solved values of a , b , and c are in agreement with the values specified in line 4 of the listing. Note that, due to the nature of the equations in (5.10), that order does not matter.

Maximum Likelihood Estimation (MLE)

Maximum likelihood estimation is another commonly used technique for creating point estimators. In fact, in the study of mathematical statistics, it is probably the most popular method used. The key idea is to consider the *likelihood* of the parameter θ given observation values, x_1, \dots, x_n . This is done via the likelihood function, which is presented below for the i.i.d. case of continuous probability distributions,

$$L(\theta ; x_1, \dots, x_n) = f_{X_1, \dots, X_n}(x_1, \dots, x_n ; \theta) = \prod_{i=1}^n f(x_i ; \theta). \quad (5.11)$$

In the second equality, the joint probability density of X_1, \dots, X_n is represented as the product of the individual probability densities, since the observations are assumed i.i.d.

A key observation is that the likelihood, $L(\cdot)$, in (5.11) is a function of the parameter θ , influenced by the sample, x_1, \dots, x_n . Now given the likelihood, the *maximum likelihood estimator* is a value $\hat{\theta}$ that maximizes $L(\theta ; x_1, \dots, x_n)$. The rational behind using this as an estimator is that it chooses the parameter value θ that is most plausible, given the observed sample.

As an example, consider the continuous uniform distribution on $[0, \theta]$. In this case, it is useful to consider the pdf for an individual observation as,

$$f(x ; \theta) = \frac{1}{\theta} \mathbf{1}\{x \in [0, \theta]\}, \quad \text{for } x \in \mathbb{R}.$$

Here the *indicator function*, $\mathbf{1}\{\cdot\}$ explicitly constrains the support of the random variable to $[0, \theta]$. Now using (5.11), it follows that,

$$L(\theta ; x_1, \dots, x_n) = \frac{1}{\theta^n} \prod_{i=1}^n \mathbf{1}\{x_i \in [0, \theta]\} = \frac{1}{\theta^n} \mathbf{1}\{0 \leq \min_i x_i\} \mathbf{1}\{\max_i x_i \leq \theta\}.$$

From this we see that for any sample x_1, \dots, x_n with non-negative values, this function (of θ) is maximized at $\hat{\theta} = \max_i x_i$. Hence as you can see the MLE for this case is exactly $\hat{\theta}_1$ from (5.4).

Many textbooks present constructed examples of MLEs, where the likelihood is a differentiable function of θ . In such cases, these MLEs can be solved explicitly, by carrying out the optimization of the likelihood function analytically (for example, see [CB01] and [DS11]). However, this is not always possible, and often numerical optimization of the likelihood function is carried out instead.

As an example, consider the case where we have n random samples from what we know to be a gamma distribution, with PDF,

$$f(x) = \frac{\lambda^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\lambda x}.$$

and parameters, $\lambda > 0$ and $\alpha > 0$. Since λ and α are both unknown, there is not an explicit solution to the MLE optimization problem, and hence we resort to numerical methods instead. In Listing 5.9 below, we use MLE to construct a plot of the likelihood function. That is, given our data, we calculate the Likelihood function for various combinations of α and β . Note that directly after this example, we then present an elegant approach to this numerical problem.

Listing 5.9: The likelihood function for a gamma distributions parameters

```

1  using Random, Distributions, PyPlot
2  Random.seed!(0)
3
4  actualAlpha, actualLambda = 2,3
5  gammaDist = Gamma(actualAlpha, 1/actualLambda)
6  n = 10^3
7  sample = rand(gammaDist, n)
8
9  alphaGrid = 1.5:0.01:2.5
10 lambdaGrid = 2:0.01:3.5
11
12 likelihood = [ prod([pdf(Gamma(a,1/l),v) for v in sample])
13                      for a in alphaGrid, l in lambdaGrid]
14 plot_surface(lambdaGrid, alphaGrid, likelihood, rstride=1,
15                  cstride=1, linewidth=0, antialiased=false, cmap="coolwarm");

```

- In lines 4-5 we specify the parameters α and λ , as well as the underlying distribution, `gammaDist`. Note that the gamma distribution in Julia, `Gamma()`, uses a different parameterization to what is outlined in 3 (i.e. `Gamma()` uses α , and $1/\lambda$).

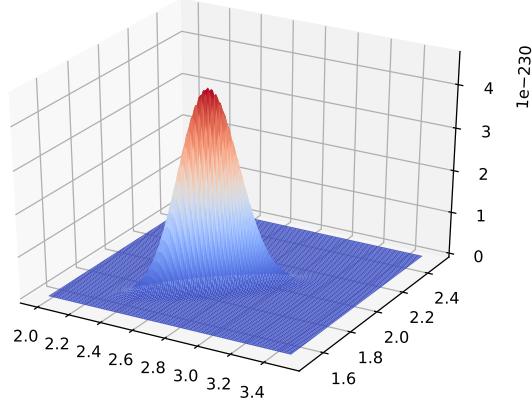


Figure 5.8: Likelihood function on different combinations of α and λ for a gamma distribution.

- In lines 6-7, we generate n sample observations, `sample`.
- In lines 9-10 we specify the grid of values over which we will calculate the likelihood function, based on various combinations of α and λ .
- In lines 12-13 we first evaluate the likelihood function (equation (5.11)), through the use of the `prod` function on an array of all pdf values, evaluated for each sample observation, `v`. Through the use of a two-way comprehension, this process is repeated for all possible combinations of `a` and `l` in `alphaGrid` and `lambdaGrid` respectively. This results in a 2-dimensional array of evaluated likelihood functions for various combinations of α and λ , denoted `likelihood`.
- In lines 14-15, the `plot_surface` function from the `PyPlot` package is used to plot the values in `likelihood` along with the corresponding values in `alphaGrid` and `lambdaGrid`.
- Note each sample, x_1, \dots, x_n generates its own likelihood function.

Having performed MLE by numerically evaluating various combinations of input parameters in the example above, we now investigate this optimization problem further, and in the process present further insight into Maximum Likelihood Estimation.

First observe that any maximizer, $\hat{\theta}$, of $L(\theta ; x_1, \dots, x_n)$ will also maximize its logarithm. Practically, (both in analytic and numerical cases), considering this *log-likelihood function* is often more attractive:

$$\ell(\theta ; x_1, \dots, x_n) := \log L(\theta ; x_1, \dots, x_n) = \sum_{i=1}^n \log(f(x_i ; \theta)).$$

Hence, given a sample from a gamma distribution as before, the log-likelihood function is,

$$\ell(\theta ; x_1, \dots, x_n) = n\alpha \log(\lambda) - n \log(\Gamma(\alpha)) + (\alpha - 1) \sum_{i=1}^n \log(x_i) - \lambda \sum_{i=1}^n x_i.$$

We may then divide by n (without compromising the optimizer) to obtain the following function that needs to be maximized:

$$\tilde{\ell}(\theta ; \bar{x}, \bar{x}_\ell) = \alpha \log(\lambda) - \log(\Gamma(\alpha)) + (\alpha - 1)\bar{x}_\ell - \lambda\bar{x}.$$

where, \bar{x} is the sample mean and,

$$\bar{x}_\ell := \frac{1}{n} \sum_{i=1}^n \log(x_i).$$

Further simplification is possible by removing the stand-alone $-\bar{x}_\ell$ term, as it does not affect the optimal value. Hence, our optimization problem is then,

$$\max_{\lambda > 0, \alpha > 0} \alpha(\log(\lambda) + \bar{x}_\ell) - \log(\Gamma(\alpha)) - \lambda\bar{x}. \quad (5.12)$$

As is typical in such cases, the function actually depends on the sample, only through the two *sufficient statistics*, \bar{x} and \bar{x}_ℓ . Now in optimizing (5.12), we aren't able to obtain an explicit expression for the maximizer. However, taking α as fixed, we may consider the derivative with respect to λ , and equate this to 0:

$$\frac{\alpha}{\lambda} - \bar{x} = 0.$$

Hence, for any optimal α^* , we have that $\lambda^* = \alpha^*/\bar{x}$. This allows us to substitute λ^* for λ in (5.12) to obtain:

$$\max_{\alpha > 0} \alpha(\log(\alpha) - \log(\bar{x}) + \bar{x}_\ell) - \log(\Gamma(\alpha)) - \alpha. \quad (5.13)$$

Now by taking the derivative of (5.13) with respect to α , and equating this to 0, we obtain,

$$\log(\alpha) - \log(\bar{x}) + \bar{x}_\ell + 1 - \psi(\alpha) - 1 = 0,$$

where $\psi(z) := \frac{d}{dz} \log(\Gamma(z))$ is the well known *digamma function*. Hence we find that α^* must satisfy:

$$\log(\alpha) - \psi(\alpha) - \log(\bar{x}) + \bar{x}_\ell = 0. \quad (5.14)$$

In addition, since $\lambda^* = \alpha^*/\bar{x}$, our optimal MLE solution is given by (λ^*, α^*) . In order to find this value, (5.14) must be solved numerically.

In Listing 5.10 below we do just this. In fact, we repeat the act of numerically solving (5.14) many times, and in the process illustrate the distribution of the MLE in terms of λ and α . Note that there are many more properties of the MLE that we do not discuss here, including the asymptotic distribution of the MLE, which happens to be a multi-variate normal. However, through this example, we provide an intuitive illustration of the distribution of the MLE, which is bivariate in this case, and can be observed in Figure 5.9.

Listing 5.10: MLE of a gamma distributions parameters

```

1  using SpecialFunctions, Distributions, Roots, PyPlot
2
3  eq(alpha, xb, xbl) = log.(alpha) - digamma.(alpha) - log(xb) + xbl
4
5  actualAlpha, actualLambda = 2, 3
6  gammaDist = Gamma(actualAlpha, 1/actualLambda)
7
8  function mle(sample)
9      alpha = find_zero( (a)->eq(a, mean(log.(sample))), 1)

```

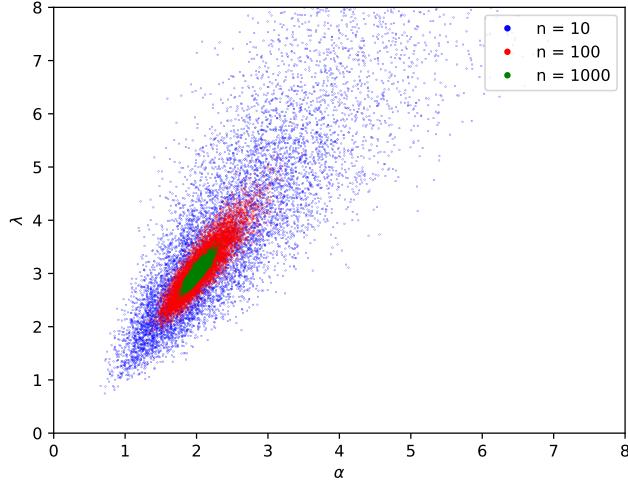


Figure 5.9: Repetitions of MLE estimates of a $\text{gamma}(2, 3)$ distribution with $n = 10, 100, 1000$. For $n = 1000$ the asymptotic normality of the MLE is visible.

```

10     lambda = alpha/mean(sample)
11     return [alpha,lambda]
12 end
13
14 N = 10^4
15
16 mles10 = [mle(rand(gammaDist,10)) for _ in 1:N]
17 mles100 = [mle(rand(gammaDist,100)) for _ in 1:N]
18 mles1000 = [mle(rand(gammaDist,1000)) for _ in 1:N]
19
20 plot(first.(mles10),last.(mles10),"b.",ms="0.3",label="n = 10")
21 plot(first.(mles100),last.(mles100),"r.",ms="0.3",label="n = 100")
22 plot(first.(mles1000),last.(mles1000),"g.",ms="0.3",label="n = 1000")
23 xlabel(L"\alpha")
24 ylabel(L"\lambda")
25 xlim(0,8)
26 ylim(0,8)
27 legend(markerscale=20,loc="upper right")

```

- In line 1, we include the `SpecialFunctions` and `Roots` packages, as they contain the `digamma()` and `find_zeros()` functions respectively, which are used later in this example.
- In line 3, the `eq()` function implements equation (5.14). Note it takes three arguments, an `alpha` value `alpha`, a sample mean `xb`, and the mean of the log of each observation `xbl`, which is calculated element wise via `log().`. This allows us to apply `eq()` on vectors.
- In lines 5-6 we specify the actual parameters of the underlying gamma distribution, as well as the distribution itself.
- In lines 8-12 the function `mle()` is defined, which in line 9 takes an array of sample observations, and solves the value of `alpha` which satisfies the zero of `eq()`. This is done through

the use of the `find_zero()` function, and the anonymous function

```
(a) ->eq(a, mean(sample), mean(log.(sample)))
```

Note also the trailing 1 in line 9, which is used as the initial value of the iterative solver. In line 10, the corresponding `lambda` value is calculated, and both `alpha` and `lambda` are returned as an array of values.

- In line 16, 10 random samples are made from our gamma distribution, and then the function `mle()` is used to solve for the corresponding values of `alpha` and `lambda`. and an array of arrays This experiment is repeated through a comprehension `N` times total, and the resulting array of arrays stored as `mles10`.
- Lines 17-18 repeat the same procedure as that in line 16, however in these two cases, the experiments are conducted for 100 and 1000 random samples respectively.
- In lines 20-22, a scatterplot of the resulting pairs of $\hat{\alpha}$ and $\hat{\lambda}$ are plotted, for the cases of the sample size being equal to 10, 100 and 1000. Note the use of the `first()` and `last()` functions, which are used to return the values of `alpha` and `lambda` respectively.
- Note that the bi-variate distribution of `alpha` and `lambda` can be observed. In addition, for a larger number of observations, it can be seen that the data is centered about the true underlying parameters `alpha` and `lambda` values of 2 and 3. This agrees with the fact that the MLE is asymptotically unbiased.

Comparing the Method of Moments and MLE

We now carry out an illustrative comparison between a method of moments estimator and an MLE estimator on a specific example. Consider a random sample x_1, \dots, x_n , which has come from a uniform distribution on the interval (a, b) . The MLE for the parameter $\theta = (a, b)$, can be shown to be,

$$\hat{a} = \min\{x_1, \dots, x_n\}, \quad \hat{b} = \max\{x_1, \dots, x_n\}. \quad (5.15)$$

For the method of moments estimator, since $X \sim \text{uniform}(a, b)$, it follows that,

$$\mathbb{E}[X] = \frac{a+b}{2}, \quad \text{Var}(X) = \frac{(b-a)^2}{12}.$$

Hence, by solving for a and b , and replacing $\mathbb{E}[X]$ and $\text{Var}(X)$ with \bar{x} and s^2 respectively, we obtain,

$$\hat{a} = \bar{x} - \sqrt{3}s, \quad \hat{b} = \bar{x} + \sqrt{3}s. \quad (5.16)$$

Now we can compare how the estimators (5.15) and (5.16) perform based on MSE. Specifically, the variance and bias. In Listing 5.11 below, we use Monte Carlo simulation to compare the estimates of \hat{b} using both the method of moments and MLE, for different cases of n .

Listing 5.11: MSE, bias and variance of estimators

```

1  using Distributions, PyPlot
2
3  min_n, step_n, max_n = 10, 10, 100
4  sampleSizes = min_n:step_n:max_n
5
```

```

6  MSE = Array{Float64}(undef, Int(max_n/step_n), 6)
7
8  trueB = 5
9  trueDist = Uniform(-2, trueB)
10 N = 10^4
11
12 MLEest(data) = maximum(data)
13 MMest(data) = mean(data) + sqrt(3)*std(data)
14
15 for (index, n) in enumerate(sampleSizes)
16
17     mleEst = Array{Float64}(undef, N)
18     mmEst = Array{Float64}(undef, N)
19     for i in 1:N
20         sample = rand(trueDist,n)
21         mleEst[i] = MLEest(sample)
22         mmEst[i] = MMest(sample)
23     end
24     meanMLE = mean(mleEst)
25     meanMM = mean(mmEst)
26     varMLE = var(mleEst)
27     varMM = var(mmEst)
28
29     MSE[index,1] = varMLE + (meanMLE - trueB)^2
30     MSE[index,2] = varMM + (meanMM - trueB)^2
31     MSE[index,3] = varMLE
32     MSE[index,4] = varMM
33     MSE[index,5] = meanMLE - trueB
34     MSE[index,6] = meanMM - trueB
35 end
36
37 figure("MM vs MLE comparrision", figsize=(12,4))
38 subplots_adjust(wspace=0.2)
39
40 subplot(131)
41 plot(sampleSizes,MSE[:,1],"xb",label="Mean sq.err (MLE)")
42 plot(sampleSizes,MSE[:,2],"xr",label="Mean sq.err (MM)")
43 xlabel("n")
44 legend(loc="upper right")
45
46 subplot(132)
47 plot(sampleSizes,MSE[:,3],"xb",label="Variance (MLE)")
48 plot(sampleSizes,MSE[:,4],"xr",label="Variance (MM)")
49 xlabel("n")
50 legend(loc="upper right")
51
52 subplot(133)
53 plot(sampleSizes,MSE[:,5],"xb",label="Bias (MLE)")
54 plot(sampleSizes,MSE[:,6],"xr",label="Bias (MM)")
55 xlabel("n")
56 legend(loc="center right");

```

- In lines 3-4 the minimum and maximum sample size observations is specified, and with step size.
- In line 6 the two-dimensional array MSE is pre-allocated. The rows represent the different cases of the number of observations in each group. The 6 columns represent the different

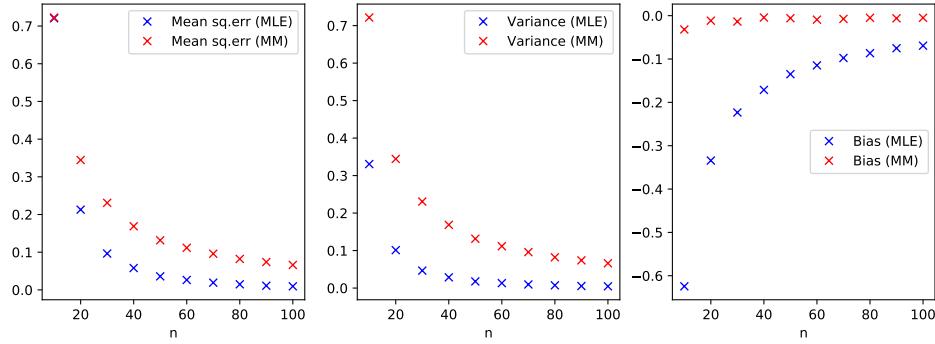


Figure 5.10: Comparing the method of moments and MLE in terms of MSE, variance, and bias.

components of the MSE as calculated via MSE and method of moments respectively. The first and second columns represent the MSE, the third and fourth represent the variance and the fifth and sixth columns the bias.

- In lines 8-9 we specify the true parameter we are trying to estimate \hat{b} , and the underlying uniform distribution `trueDist`.
- In line 10 we specify that we will repeat each experiment N times, for each case of n observations.
- In lines 12-13 equations (5.15) and (5.16) are implemented via the functions `MLEest()` and `MMest()` respectively.
- Lines 15-35 contain the core of this code example. In lines 19-23 we randomly sample from our distribution n observations, and use maximum likelihood and method of moments to estimate our parameter `trueB`. This experiment is repeated N times total, and the parameter estimates stored in the arrays `mleEst` and `mmEst` (which were pre-allocated in lines 17-18). In lines 24-27 the means and variances of both separate estimate groups is calculated, and stored in the variables `meanMLE`, `meanMM`, `varMLE` and `varMM`. These are then used in the lines 29-34, where the mean squared error, the variance, and the bias of these estimates is calculated. This process is repeated for each case of sample sizes in `sampleSizes`.
- The remaining lines are used to plot Figure 5.10. It can be observed that as the number of sample observations increase n , the mean squared error reduces. It can be observed that overall maximum likelihood estimation has a lower mean squared error and variance when compared to the method of moments. However, MLE has a higher bias than MM, which is almost zero, even for small sample sizes.

5.5 Confidence Interval as a Concept

Now that we have dealt with the concept of a point estimator, we consider how confident we are about our estimate. The previous section included analysis of such confidence in terms of the mean squared error and its variance and bias components. However, given a single sample, X_1, \dots, X_n ,

how does one obtain an indication about the accuracy of the estimate? Here the concept of a *confidence interval* comes as an aid.

Consider the case where we are trying to estimate the parameter θ . A confidence interval is then an interval $[L, U]$ obtained from our sample data, such that,

$$\mathbb{P}(L \leq \theta \leq U) = 1 - \alpha, \quad (5.17)$$

where $1 - \alpha$ is called the *confidence level*. Knowing this range $[L, U]$ in addition to θ is useful, as it indicates some level of certainty in regards to the unknown value.

Much of elementary classical statistics involves explicit formulas for L and U , based on the sample X_1, \dots, X_n . Most of Chapter 6 is dedicated to this, however in this section we simply introduce the concept through an elementary non-standard example. Consider a case of a single observation ($n = 1$) taken from a symmetric triangular distribution, with a spread of 2 and an unknown center (mean) μ . In this case, we would set,

$$L = X + q_{\alpha/2}, \quad U = X + q_{1-\alpha/2},$$

where q_u is the u 'th quantile of a triangular distribution centered at 0, and having a spread of 2. Note that this is not the only possible construction of a confidence interval, however it makes sense due to the symmetry of the problem. It is easy to see that $q_{\alpha/2} = -1 + \sqrt{\alpha}$ and $q_{1-\alpha/2} = 1 - \sqrt{\alpha}$.

Now, given observations, (a single observation in this case), we can compute, L and U . This is performed in Listing 5.12 below.

Listing 5.12: A confidence interval for a symmetric triangular distribution

```

1  using Random, Distributions
2  Random.seed!(0)
3
4  alpha = 0.05
5  q=quantile(TriangularDist(-1,1,0),1-alpha/2)
6  L(obs) = obs - (1-sqrt(alpha))
7  U(obs) = obs + (1-sqrt(alpha))
8
9  mu=5.57
10 observation = rand(TriangularDist(mu-1,mu+1,mu))
11 println("Lower bound L: ", L(observation))
12 println("Upper bound U: ", U(observation))

```

```

Lower bound L: 5.1997170907797585
Upper bound U: 6.7525034952798

```

- In line 5 we set the quantile value q , based on a symmetric triangular distribution with spread 2, based value of α specified in line 3.
- In lines 6-7, the functions $L()$ and $U()$ implement the formulas above.
- In this simple example, the estimate of the mean, μ is simply the singe observation, given in line 9.

- The virtue of the example is in presenting the 95% confidence interval, as output by lines 11 and 12. Hence in this example, we know that with probability 0.95, the unknown parameter lies in [5.2, 6.75].

Let us now further explore the meaning of a confidence interval by considering (5.17). The key point is that there is a $1 - \alpha$ chance that the actual parameter θ lies in the interval $[L, U]$. This means that if the sampling experiment is repeated say N times, then on average, $100 \times (1 - \alpha)\%$ of the times the actual parameter θ is covered by the interval.

In Listing 5.13 below, we present an example where we repeat the sampling process N times; where each time we take a single sample (a single observation in this case) and construct the corresponding confidence interval. We observe that only about $\alpha/100$ times the confidence interval, $[L, U]$, does not include the parameter in question, μ .

Listing 5.13: A simple CI in practice

```

1  using Random, Distributions, PyPlot
2  Random.seed!(2)
3
4  mu = 5.57
5  alpha = 0.05
6  L(obs) = obs - (1-sqrt(alpha))
7  U(obs) = obs + (1-sqrt(alpha))
8
9  tDist = TriangularDist(mu-1,mu+1,mu)
10 N = 100
11
12 for k in 1:N
13     observation = rand(tDist)
14     LL,UU = L(observation), U(observation)
15     plt[:bar](k, bottom = LL, UU-LL, color= (LL < mu && mu < UU) ? "b" : "r")
16 end
17
18 plot([0,N+1], [mu,mu], c="k", label="Parameter value")
19 legend(loc="upper right")
20 ylabel("Value")
21 xticks([])
22 xlim(0,N+1)
23 ylim(3,8)
```

- Lines 4-7 are similar to those from Listing 5.12 previously. In these lines we define the underlying parameter `mu`, as well as set the value of `alpha`, and define the functions `L()` and `U` as before.
- In line 9 we specify our underlying triangular distribution
- In line 10 we specify that we will generate N confidence intervals.
- In lines 12-16, we generate our confidence intervals. In line 12 we make a single sample observation from `tDist`, and then compute the lower and upper limits of our confidence interval, `LL` and `UU` respectively. In line 15, the confidence interval is plotted as a bar chart, and is color coded using a logical statement. If the confidence interval contains the parameter

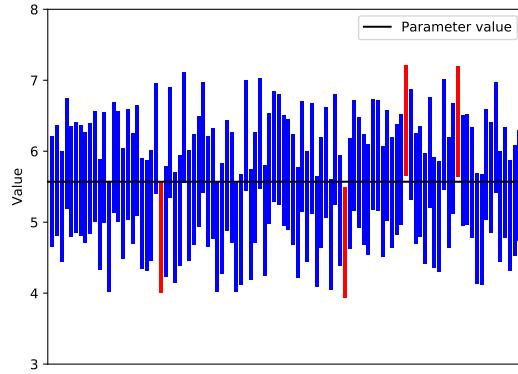


Figure 5.11: 100 confidence intervals. The blue confidence interval bars contain the unknown parameter, while the red ones do not.

μ , then the bar is colored blue, else it is colored red. This process is repeated N times, generating N confidence intervals total.

- Line 17 plots the value of the parameter as a horizontal line in black. It can be observed that out of the 100 confidence intervals generated, only 4 of them do not contain our parameter μ . This is what we expect, as we specified an α value of 0.05, i.e we are 95% confident that each of these confidence intervals contains our parameter.

5.6 Hypothesis Tests Concepts

Having explored point estimation and confidence intervals, we now consider ideas associated with *Hypothesis Testing*. The approach involves partitioning the parameter space Θ , into Θ_0 and Θ_1 , and then, based on the sample, concluding whether one of two hypotheses, H_0 or H_1 , hold, where,

$$H_0 : \theta \in \Theta_0, \quad H_1 : \theta \in \Theta_1. \quad (5.18)$$

The hypothesis H_0 is called the *Null Hypothesis* and H_1 the *Alternative Hypothesis*. The former is the default hypothesis, and in carrying out hypothesis testing, our general aim (or hope) is to reject this, with a guaranteed low probability, α .

		Decision	
		Do not reject H_0	Reject H_0
Reality	H_0 is true	Correct ($1-\alpha$)	Type I error (α)
	H_0 is false	Type II error (β)	Correct ($1-\beta$)

Table 5.1: Type I and Type II errors with their probabilities α and β respectively.

Since we estimate, θ by $\hat{\theta}$ based on a random sample, there is always a chance of making a

mistaken false conclusion. As summarized in Table 5.1, the two errors that can be made are a *Type I Error*: Rejecting H_0 falsely, or a *Type II Error*: Failing to correctly reject H_0 . The probability α quantifies the likelihood of making a Type-I error, while the probability of making a Type-II error is denoted by β . Note that $1 - \beta$ is known as the *Power* of the hypothesis test, and the concept of power is covered in more detail in Section 7.5. Note that in carrying out a hypothesis test, α is typically specified, while power is left uncontrolled.

We now present some elementary examples which illustrate the basic concepts involved. Note that standard hypothesis tests are discussed in depth in Chapter 7.

Simple Hypothesis Tests

When the alternative spaces, Θ_0 and Θ_1 are only comprised of a single point each, the hypothesis test is called a *Simple Hypothesis Test*. These tests are often not of great practical use, but are introduced for pedagogical purposes. Specifically, by analyzing such tests we can understand how Type I errors and Type II errors interplay.

As an introductory example, consider a container that contains two identical types of bolts, except that one type weights 15 grams on average and the other 18 grams on average. The standard deviation of the weights of both bolt types is 2 grams. Imagine now that we sample a single bolt, and wish to determine its type. Denote the weight of this bolt by the random variable X . For this example, we devise the following statistical hypothesis test: $\Theta_0 = \{15\}$ and $\Theta_1 = \{18\}$. Now, given a threshold τ , we reject H_0 if $X > \tau$, otherwise we retain H_0 .

In this circumstance, we can explicitly analyze the probabilities of both the Type I and Type II errors. Listing 5.14 below generates Figure 5.12, which illustrates this graphically for $\tau = 17.5$.

Listing 5.14: A simple hypothesis test

```

1  using Distributions, StatsBase, PyCall, PyPlot
2  @pyimport matplotlib.patches as patch
3
4  mu0, mu1, sd  = 15, 18, 2
5  tau = 17.5
6  dist0 = Normal(mu0,sd)
7  dist1 = Normal(mu1,sd)
8
9  gridMin, gridMax = 5, 25
10 grid = gridMin:0.1:gridMax
11
12 fig = figure(figsize=(6,6))
13 ax = fig[:add_subplot](1,1,1)
14 plot(grid,pdf(dist0,grid),"b", label="Bolt type 15g")
15 plot(grid,pdf(dist1,grid),"g", label="Bolt type 18g")
16 plot([tau,gridMax],[0,0],"r",lw=3,label="Rejection region")
17
18 verts1 = [ [[tau,0.0]]; [[i, pdf(dist0,i)] for i in tau:0.1:gridMax] ;
19           [[gridMax,0.0]] ]
20 poly1 = patch.Polygon(verts1, fc="blue", ec="0.5",alpha=0.2)
21
22 verts2 = [ [[gridMin,0.0]]; [[i, pdf(dist1,i)] for i in gridMin:0.1:tau] ;
23           [[tau,0.0]] ]
24 poly2 = patch.Polygon(verts2, fc="green", ec="0.5",alpha=0.2)
```

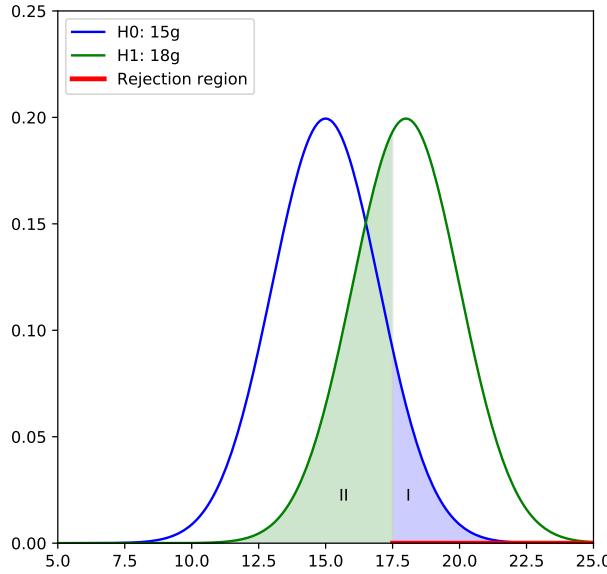


Figure 5.12: Type I (blue) and Type II (green) errors. The rejection region is colored with red on the horizontal axis.

```

24      [[tau,0.0]] ]
25  poly2 = patch.Polygon(verts2, fc="green", ec="0.5", alpha=0.2)
26  ax[:add_artist](poly1)
27  ax[:add_artist](poly2)
28
29 xlim(gridMin,gridMax)
30 ylim(0,0.25)
31 legend(loc="upper left")
32
33 println("Probability of Type I error: ", ccdf(dist0,tau))
34 println("Probability of Type II error: ", cdf(dist1,tau))

```

```

Probability of Type I error: 0.10564977366685525
Probability of Type II error: 0.4012936743170763

```

- In line 1-2 the PyCall function is called, as is required for the patches object, which is called via the @pyimport command.
- In line 4 the parameters of our two distributions are set.
- In line 5 we set the threshold for which we perform our hypothesis, tau.
- In lines 6-7, we create the two distributions for our two different bolt types.
- In lines 9-10, we set the minimum and maximum values of our grid, as well as the grid itself. These values are used later in shading areas of our plot.

- In lines 15-17 the two distributions for H_0 and H_1 are plotted, along with the rejection region (in red).
- In lines 19-27 the regions corresponding to the probability of making a Type I and Type II error are shaded in blue and green respectively. This is done through the use of Matlab patches.
- In lines 33-34 the probabilities of making a type I and type II error are calculated through the use of the `ccdf()` and `cdf` functions respectively.

The Test Statistic, p -value, and Rejection Region

We now introduce several important concepts in hypothesis testing, the *test statistic*, the *p -value*, and the *rejection region*. Once the hypothesis tests setup has been established by partitioning the parameter space according to (5.18), the next step is to calculate the test statistic from the sample data. Importantly, the test statistic is a random variable itself.

Since the test statistic follows some distribution under H_0 , the next step is to consider how likely it is that we observe the test statistic calculated from our sample data. To this end, in setting up the hypothesis test we typically choose a significance level, α . It typically between 0.05 and 0.01.

This α is the proportion of the rejection region, and can be thought of as the area corresponding to the outermost fraction of the distribution of the test statistic under H_0 . That is, under H_0 , we expect to observe the test statistic in this area only a fraction α of the time. Hence, if we observe the test statistic in this region, we reject the null hypothesis, and say that “there is sufficient evidence to reject H_0 at the α significance level”. If however, the test statistic does not fall within the rejection region, then we fail to reject H_0 .

An associated concept is the p -value, which is simply the probability (under H_0) of observing a test statistic equal to, or more ‘extreme’ than the one calculated. Hence often one will speak in terms of p -value, and if the p -value is less than 0.05 (for example), then we reject the null hypothesis in favor of the alternative (assuming $\alpha = 0.05$).

We now present a simple yet non-standard example to help reinforce these concepts and how they relate. For this example, consider that we have a series of sample observations from some random variable, X , that is distributed continuous uniform between 0 and some unknown upper bound, m .

Say that we set,

$$\begin{aligned} H_0 : m &= 1, \\ H_1 : m &\neq 1. \end{aligned}$$

One possible test statistic for such as test is the sample range:

$$R = \max(x_1, \dots, x_n) - \min(x_1, \dots, x_n).$$

As is always the case, the test statistic is a random variable, and although the distribution of R is not immediately obvious, we use Monte Carlo simulation to approximate it. We then find the

α^{th} quantile of the empirical data, identify the rejection region, and finally use these to make some conclusion about H_0 . We perform the above in Listing 5.15, and discuss the resulting output shown below.

Listing 5.15: The distribution of a test statistic under H_0

```

1  using Random, Statistics, PyPlot
2  Random.seed!(2)
3
4  N = 10^7
5  n = 10
6  alpha = 0.05
7
8  function ts(n)
9      sample = rand(n)
10     return maximum(sample)-minimum(sample)
11 end
12
13 empiricalDistUnderH0 = [ts(10) for _ in 1:N]
14 rejectionValue = quantile(empiricalDistUnderH0,alpha)
15
16 sample = 0.75*rand(n)
17 testStat = maximum(sample)-minimum(sample)
18 pValue = sum(empiricalDistUnderH0 .<= testStat)/N
19
20 if testStat > rejectionValue
21     print("Didn't reject: ", round(testStat,digits=4))
22     print(" > ", round(rejectionValue,digits=4))
23 else
24     print("Reject: ", round(testStat,digits=4))
25     print(" <= ", round(rejectionValue,digits=4))
26 end
27 println("\np-value = $(round(pValue,digits=4))")
28
29 plt[:hist](empiricalDistUnderH0,100, color="b", histtype="step", normed="true")
30 plot([testStat, testStat],[0,4],"r", label="Observed test \nstatistic")
31 plot([rejectionValue, rejectionValue],[0,4],"k--",
32       label="Critical value \nboundary")
33 legend(loc="upper left")
34 ylim(0,4)

```

```

Reject: 0.517 <= 0.6058
p-value = 0.0141

```

- The test statistic for the range will be max of sample - min of sample.
- We don't know the distribution of the test statistic, so we empirically create this via Monte Carlo.
- Note that for this example, our sample data is simply $0.75 * \text{rand}(n)$.
- We then again empirically calculate the p -value corresponding to our test statistic by counting the proportion of observations that are less than or equal to testStat.

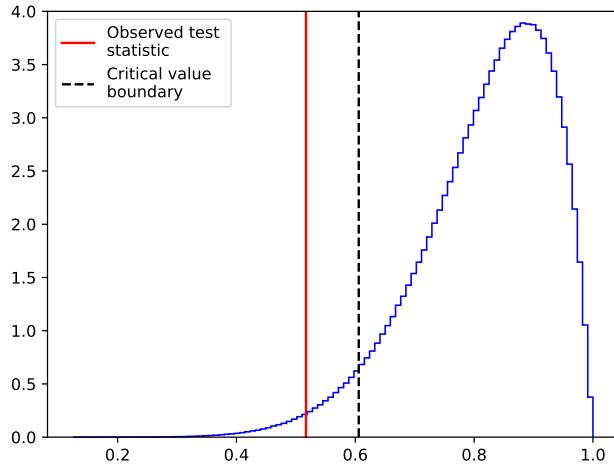


Figure 5.13: The distribution of the test statistic, R , under H_0 . With $\alpha = 0.05$ the rejection region is to the left of the black dashed line. In a specific sample, the test statistic is on the red line and we reject H_0 .

A Randomized Hypothesis Test

We now investigate the concept of a *randomization test*, which is a type of *non-parametric test*, i.e. a statistical test which does not require that we know what type of distribution the data comes from. A virtue of non-parametric tests is that they do not impose a specific model.

Consider the following example, where a farmer wants to test whether a new fertilizer is effective at increasing the yield of her tomato plants. As an experiment, she took 20 plants, kept 10 as controls and treated the remaining 10 with fertilizer. After two months, she harvested the plants, and recorded the yield of each plant (in kg) as shown in Table 5.2.

Control	4.17	5.58	5.18	6.11	4.5	4.61	5.17	4.53	5.33	5.14
Fertilizer	6.31	5.12	5.54	5.5	5.37	5.29	4.92	6.15	5.8	5.26

Table 5.2: Yield in kg for 10 plants with, and 10 plants without fertilizer (control).

It can be observed that the group of plants treated with fertilizer have an average yield 0.494 kg greater than that of the control group. One could argue that this difference is due to the effects of fertilizer. We now investigate if this is a reasonable assumption.

Let us assume for a moment that the fertilizer had no effect on plant yield, and that the result was simply due to random chance. In such a scenario, we actually have 20 observations from the same group, and regardless of how we group our observations we would expect to observe similar results.

Hence we can investigate the likelihood of this outcome occurring by random chance, by considering all possible *combinations* of 10 samples from our group of 20 observations, and counting

how many of these combinations result in a difference in sample means greater than 0.494 kg. The proportion of times this occurs is analogous to the likelihood that the difference we observe in our sample means was purely due to random chance.

First we calculate the number of ways one can sample $r = 10$ unique items from $n = 20$ total, which is given by,

$$\binom{20}{10} = 184,756.$$

Hence the number of possible combinations in our example is computationally manageable. In Listing 5.16 below, we use Julia's Combinatorics package to enumerate the difference in sample means for every possible combination. From the output below we observe that only 2.39% of all possible combinations result in a sample mean greater than or equal to our treated group (i.e. a difference greater or equal to 0.494 kg). Therefore there is significant statistical evidence that the fertilizer increases the yield of the tomato plants, since under H_0 , there is only a 2.39% chance of obtaining this value or better by random chance.

Listing 5.16: A randomized hypothesis test

```

1  using Combinatorics, DataFrames, CSV
2
3  data = CSV.read("fertilizer.csv")
4  control = data.Control
5  fertilizer = data.FertilizerX
6
7  x = collect(combinations([control;fertilizer],10))
8
9  meanFert = mean(fertilizer)
10 proportion = sum([mean(i) >= meanFert for i in x])/length(x)

```

0.023972157873086666

- In line 1 the Combinatorics package is called, as it contains the combinations() function which we use later in this example.
- In line 3-5 we import our data, and store the data for the control and fertilized groups in the arrays control and fertilizer.
- In line 7 all observations are concatenated into one array via the use of [;]. Following this, the combinations() function is used to generate an iterator object for all combinations of 10 elements from our 20 observations. The collect() function then converts this iterator into an array of all possible combinations of 10 objects, sampled from 20 total. This array of all combinations is stored as x.
- In line 9, the mean of the fertilizer group is calculated and assigned to the variable meanFert.
- In line 10 the mean of each combination in the array x is calculated and compared against meanFert. The proportion of means which are greater than or equal to meanFert is then calculated through the use of a comprehension, and the functions sum() and length().
- It can be seen that approximately 2.4% of all combinations result in a sample mean greater than the fertilized group, hence there is statistical significance that the fertilizer increases tomato plant yield.

The Receiver Operating Curve

In the previous example, $\tau = 17.5$ was arbitrarily chosen. Clearly, if τ was increased, the probability of making a Type I error would decrease, while the probability of making a type II error would increase. Conversely, if we decreased τ , the reverse would occur. We now introduce the *Receiver Operating Curve* (ROC), which is a tool that helps us to visualize the tradeoff between Type I and Type II errors simultaneously. It allows us to visualize the error tradeoffs for all possible, τ simultaneously, for a particular alternative hypothesis H_1 . ROC's are also a way of comparing different sets of hypothesis simultaneously. Despite their usefulness, they are a concept not often taught in elementary statistics courses.

We now present an example of the ROC by considering our previous screw classification problem, but this time we look at three different scenarios for $\mu_1 : 16, 18$ and 20 simultaneously. Clearly, the bigger the difference between μ_0 and μ_1 , the easier it should be to classify the screw without making errors.

In Listing 5.17 below, we consider each scenario and shift τ , and in the process plot the analytic coordinates of $(\alpha(\tau), 1 - \beta(\tau))$ (where α and β are the probabilities of making a Type 1 and Type II error respectively). This results in ROC plots shown in Figure 5.14. To help visualize how the ROCs are generated, one can consider Figure 5.12 and imagine the effect of sliding τ . By plotting several different ROCs on the same figure, we can compare the likelihood of making errors for various scenarios of different μ_1 's.

Listing 5.17: Comparing receiver operating curves

```

1  using Distributions, StatsBase, PyPlot
2
3  mu0, mula, mulb, mulc, std = 15, 16, 18, 20, 2
4
5  dist0 = Normal(mu0, std)
6  dist1a = Normal(mula, std)
7  dist1b = Normal(mulb, std)
8  dist1c = Normal(mulc, std)
9
10 tauGrid = 5:0.1:25
11
12 falsePositive = ccdf(dist0,tauGrid)
13 truePositiveA = ccdf(dist1a,tauGrid)
14 truePositiveB = ccdf(dist1b,tauGrid)
15 truePositiveC = ccdf(dist1c,tauGrid)
16
17 figure("ROC",figsize=(6, 6))
18 subplot(111)
19 plot([0,1],[0,1],"--k", label="H0 = H1 = 15")
20 plot(falsePositive, truePositiveA, "b", label=L"H1a: $\mu_1$ = 16")
21 plot(falsePositive, truePositiveB, "r", label=L"H1b: $\mu_1$ = 18")
22 plot(falsePositive, truePositiveC, "g", label=L"H1c: $\mu_1$ = 20");
23
24 PyPlot.legend()
25 xlim(0,1)
26 ylim(0,1)
27 xlabel(L"\alpha")
28 ylabel("Power")

```

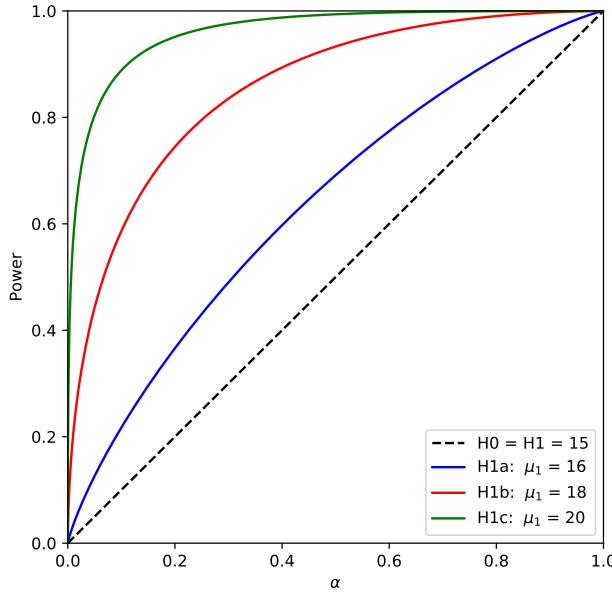


Figure 5.14: Three ROCs for various points within H_1 .

As the difference between the point in H_1 and H_0 increases, the likelihood of making a an error reduces.

- In line 3 the parameters of our four different distributions are defined. The standard deviation for each is the same, but the means of H_0 , and our three separate H_1 's (H_{1a} , H_{1b} , H_{1c}) are different.
- In lines 5-8, we define four distribution objects using the parameters from line 3.
- In line 10, we define the grid of values of τ over which we will evaluate the likelihood of correctly rejecting H_0 (true positive, $1-\beta$, or power), against the likelihood of incorrectly rejecting H_0 (false positive, α , or Type I error).
- In line 12 we calculate the probability of incorrectly rejecting H_0 through the use of the `ccdf()` function.
- In lines 13-15, the power (i.e. true positive) is calculated for different thresholds of τ for each of our different hypotheses. As before in line 12, the `ccdf()` function is used. Importantly, recall that $\text{power} = 1 - \beta$, where β is the probability of making a Type II error.
- In line 19, a diagonal dashed line is plotted. This line represents the extreme case of the distributions of H_0 and H_1 directly overlapping. In this case, the probability of a Type I error the same as the power, (or $1-\beta$).
- Lines 20-22 plot the two pairs of values (false positive, and true positive) against each other for various cases of H_1 , given that $H_0 : \mu = 15$.
- To help visualize the behavior of the ROC more easily, consider it along with Figure 5.12, specifically the case of H_{1b} . In this figure, the shaded blue area represents α , while 1 minus

the green area represents power. If one considers $\tau = 25$, then both α and power are almost zero, and this corresponds (approximately) to $(0,0)$ in Figure 5.14. Now, as the threshold is slowly decreased, it can be seen that the power increases at a much faster rate than α , and this behavior is observed in the ROC, as we slide vertically up the red line for the case of H_{1b} . In addition, as the difference in means between the null and alternative hypotheses are greater, the ROC curves are shown to be pushed perpendicularly “further out” from the diagonal dashed line, reflecting the fact that the hypothesis test is more powerful, (i.e. we are more likely to correctly reject the null hypothesis), and that it is less likely to incorrectly reject the null hypothesis.

5.7 A Taste of Bayesian Statistics

In this section we now briefly explore the Bayesian approach to statistical inference as an alternative to the frequentist view of statistics which was introduced in Sections 5.4, 5.5 and 5.6, and used throughout the remainder of the book. In the Bayesian paradigm, the (scalar or vector) parameter, θ is not assumed to exist as some fixed unknown quantity, but instead is assumed to follow a distribution. That is, the parameter itself is a random variable, and the act of *Bayesian inference* is the process of obtaining more information about the distribution of θ . Such a setup is useful in many practical situations since it allows one to capture prior beliefs about the parameter, before observations are analyzed. It also allows one to carry out repeated inference in a very natural manner by allowing inference in future periods to rely on past experience.

The key objects at play are the *prior distribution* of the parameter and the *posterior distribution* of the parameter. The former is postulated beforehand, or exists as a consequence of previous inference, while the latter captures the distribution of the parameter after observations are taken into account. The relationship between the prior and the posterior is then,

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{evidence}} \quad \text{or} \quad f(\theta | x) = \frac{f(x | \theta) \cdot f(\theta)}{\int f(x | \tilde{\theta}) f(\tilde{\theta}) d\tilde{\theta}}. \quad (5.19)$$

This is nothing but Bayes’ rule applied to densities. Here the prior distribution (density) is $f(\theta)$ and the posterior distribution (density) is $f(\theta | x)$. Observing that the denominator, known as evidence, is constant with respect to the parameter θ . This allows the equation to be written as,

$$f(\theta | x) \propto f(x | \theta) \cdot f(\theta), \quad (5.20)$$

where the symbol “ \propto ” denotes “proportional to”. Hence the posterior distribution can be easily obtained up to the normalizing constant (the evidence) by multiplying the prior with the likelihood, $f(x | \theta)$.

In general, carrying out *Bayesian inference* involves the following steps:

1. Assume some distributional model with parameters θ .
2. Use previous inference experience, elicit an expert, or make an educated guess to determine a prior distribution for the parameter, $f(\theta)$. The prior distribution might be parameterized by its own parameters, called *hyperparameters*.

3. Collect data, x and obtain the likelihood $f(x | \theta)$ based on the distributional model chosen.
4. Use the relationship (5.19) to obtain the posterior distribution of the parameters, $f(\theta | x)$. In most cases, the evidence (denominator of (5.19)) is not easily computable. Hence the posterior distribution is only available up to a normalizing constant. In some special cases the form of the posterior distribution is the same as the prior distribution. In such cases, *conjugacy* holds, the prior is called a *conjugate prior*, and the hyperparameters are updated from prior to posterior.
5. The posterior distribution can then be used to make conclusions about the model. For example, if a single specific parameter value is needed to make the model concrete, a *Bayes estimate* based on the posterior distribution, such as for example the *posterior mean*, may be computed:

$$\hat{\theta} = \int f(\theta | x) d\theta. \quad (5.21)$$

Further analyses such as obtaining *credibility intervals* (similar to confidence intervals) may also be carried out.

6. The model with $\hat{\theta}$ can then be used for making conclusions. Alternatively, a whole class of models based on the posterior distribution $f(\theta | x)$ can be used. This often goes hand in hand with simulation as one is able to generate Monte Carlo samples from the posterior distribution.

Bayesian inference has gained significant popularity over the past few decades and has evolved together with the whole field of *computational statistics*. Unless conjugacy holds, there is typically not an explicit expression for the evidence (the integral in (5.19)) and hence a computational challenge is to make use of the posterior available only up to a normalizing constant. We now elaborate on the details through variants of a very simple example in order to understand the main concepts.

A Simple Poisson Example

Consider an example where an insurance company models the number of weekly fires in a city using a Poisson distribution with parameter λ . Here λ is also the expected number of fires per week. Assume that the following data is collected over a period of eight weeks,

$$x = (x_1, \dots, x_8) = (3, 7, 0, 1, 5, 3, 6, 2).$$

Each data point indicates the number of fires per week. In this case the MLE is $\hat{\lambda} = 3.375$ simply obtained by the sample mean. Hence in a frequentist approach, after 8 weeks the distribution of the number of fires per week is modelled by a Poisson distribution with $\lambda = 3.375$. One can then obtain estimates for say, the probability of having more than 10 fires in a given week as follows:

$$\mathbb{P}(\text{fires per week} > 10) = 1 - \sum_{k=0}^{10} e^{-\lambda} \frac{\lambda^k}{k!} \approx 0.0007638. \quad (5.22)$$

However, the drawback of such an approach in estimating λ is that it didn't make use of previous information. By comparison, in a Bayesian approach the estimate would allow one to incorporate

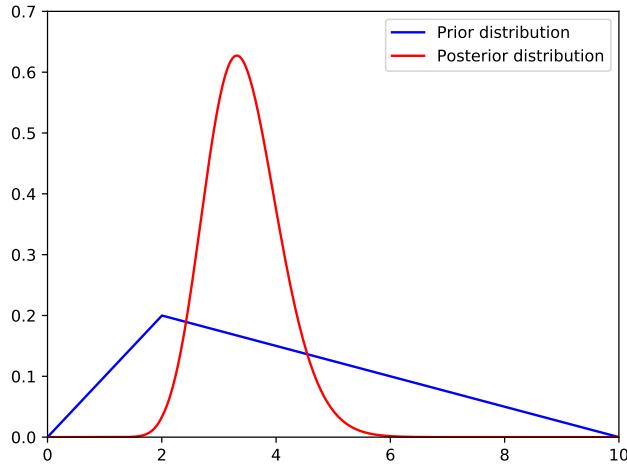


Figure 5.15: The prior distribution in blue and the posterior in red for Bayesian estimation of a Poisson distribution.

information from previous years, or alternatively from adjacent geographical areas. Say that for example, further knowledge comes to light that the number of fires per week ranges between 0 and 10 and that the typical number is 2 fires per week. In this case one can assign a prior distribution to λ that captures this belief. Here is where some critics claim that such use of Bayesian statistics turns into somewhat of a “voodoo science” since we have an infinite number of options to choose for the prior. Still, it is often useful.

In our example, assume that we decide to use a triangular distribution as shown in blue in Figure 5.15. Such a triangular distribution captures prior beliefs about the parameter λ well, because it has a defined range and a defined mode.

With the prior assigned and the data collected, we can use the machinery of Bayesian inference of (5.19). In this specific case the prior distribution of the parameter λ is the triangular distribution with the PDF,

$$f(\lambda) = \begin{cases} \frac{1}{10}x, & \lambda \in [0, 2], \\ \frac{1}{40}(10-x), & \lambda \in (2, 10]. \end{cases}$$

With the eight observations, x_1, \dots, x_8 , the likelihood is,

$$f(\lambda | x) = \prod_{k=1}^8 e^{-\lambda} \frac{\lambda^{x_k}}{x_k!}.$$

Hence the posterior is proportional to $f(\lambda | x)f(\lambda)$. However, normalization of this function in lambda requires dividing it by the evidence, given by,

$$\int_0^{10} f(x | \lambda) f(\lambda) d\lambda.$$

Typically this integral isn't easy to evaluate analytically, hence numerical methods are often used. For illustration purposes, we carry out this numerical integration as part of Listing 5.18 where we

also plot the resulting posterior distribution (red curve in Figure 5.15). To appreciate potential problems with such a numerical solution, imagine cases where the parameter θ is not just the scalar λ but rather consists of multiple dimensions. The integral of the evidence cannot be efficiently computed in such cases.

In Listing 5.18, once the prior distribution is obtained, we compute its mean to obtain a Bayes estimate for λ . The value obtained differs from the MLE obtained above and hence probability estimates using the model, such as (5.22) would also vary. Importantly, by employing the Bayesian perspective, we were able to incorporate prior knowledge into the inference procedure.

Listing 5.18: Bayesian inference with a triangular prior

```

1  using Distributions, PyPlot
2
3  prior(lam) = pdf(TriangularDist(0, 10, 2), lam)
4  data = [3, 7, 0, 1, 5, 3, 6, 2]
5
6  likelihood(lam) = *([pdf(Poisson(lam),x) for x in data]...)
7  posteriorUpToK(lam) = likelihood(lam)*prior(lam)
8
9  delta = 10^-4.
10 lamRange = 0:delta:10
11 K = sum([posteriorUpToK(lam)*delta for lam in lamRange])
12 posterior(lam) = posteriorUpToK(lam)/K
13
14 plot(lamRange, prior.(lamRange), "b", label="Prior distribution")
15 plot(lamRange, posterior.(lamRange), "r", label="Posterior distribution")
16 bayesEstimate = sum([lam*posterior(lam)*delta for lam in lamRange])
17 xlim(0, 10); ylim(0, 0.7)
18 legend(loc="upper right")

```

3.433400662486851

- In line 3 we define the prior.
- In line 4 we set the data values.
- In line 6 the likelihood function is defined. Notice that the `*` operator is used as a function, and that the splat operator, `...` is applied inside the brackets.
- Equation (5.20) is implemented in Line 7, while lines 9-11 are used to numerically compute the evidence. The actual posterior is defined in line 12.
- Lines 14 and 15 are used to plot the posterior and the prior as shown in Fig 5.15.
- In line 16 a Bayes estimate from the prior is calculated, according to (5.21).

Conjugate Priors

Following on from the previous example, a natural question arises: why use the specific form of the prior distribution that we used? After all, the results would vary if we were to choose a different prior. While in generality Bayesian statistics doesn't supply a complete answer, there are

cases where certain families of prior distributions work very well with certain (other) families of statistical models.

For example, in our case of a Poisson probability distribution model, it turns out that assuming a Gamma prior distribution works nicely. This is because the resulting posterior distribution is also guaranteed to be Gamma. In such a case, the Gamma distribution is said to be a *conjugate prior* to the Poisson distribution. The parameters of the prior/posterior distribution are called *hyperparameters*, and by exhibiting a conjugate prior distribution relationship, the hyperparameters typically have a simple update law from prior to posterior. This relieves a huge computational burden.

To see this in the case of Gamma-Poisson, assume the hyperparameters of the prior to have α (shape parameter) and β (scale parameter). Now using the Poisson likelihood and the gamma PDF we obtain:

$$\begin{aligned} \text{posterior} &\propto \left(\prod_{k=1}^n e^{-\lambda} \frac{\lambda^{x_k}}{x_k!} \right) \frac{\beta^\alpha}{\Gamma(\alpha)} \lambda^{\alpha-1} e^{-\beta\lambda} \\ &\propto e^{-n\lambda} \lambda^{\sum_{k=1}^n x_k} \lambda^{\alpha-1} e^{-\beta\lambda} \\ &= \lambda^{\alpha+\sum_{k=1}^n x_k - 1} e^{-\lambda(\beta+n)} \\ &\propto \text{gamma density with shape parameter } \alpha + \sum x_i \text{ and scale parameter } \beta + n. \end{aligned}$$

This shows us the gamma-Poisson conjugacy and implies a very neat update rule for the hyperparameters: The hyperparameter α is updated to $\alpha + \sum x_i$ and the hyperparameter β is updated to $\beta + n$.

In Listing 5.19 we use a gamma prior with posterior alpha parameters of $\alpha = 8$ and $\beta = 2$. For illustration, we compute the posterior both using the brute force method of the previous listing and using the simple hyperparameter update rule due to conjugacy. The posterior and prior are also plotted in Figure 5.16.

Listing 5.19: Bayesian inference with a gamma prior

```

1  using Distributions, PyPlot
2
3  alpha, beta = 8, 2
4  prior(lam) = pdf(Gamma(alpha, 1/beta), lam)
5  data = [3, 7, 0, 1, 5, 3, 6, 2]
6
7  likelihood(lam) = *([pdf(Poisson(lam), x) for x in data]...)
8  posteriorUpToK(lam) = likelihood(lam)*prior(lam)
9
10 delta = 10^-4.
11 lamRange = 0:delta:10
12 K = sum([posteriorUpToK(lam)*delta for lam in lamRange])
13 posterior(lam) = posteriorUpToK(lam)/K
14
15 plot(lamRange, prior.(lamRange), "b", label="Prior distribution")
16 plot(lamRange, posterior.(lamRange), "r", label="Posterior distribution")
17 bayesEstimate = sum([lam*posterior(lam)*delta for lam in lamRange])
18 xlim(0, 10); ylim(0, 0.7); legend(loc="upper right")
19

```

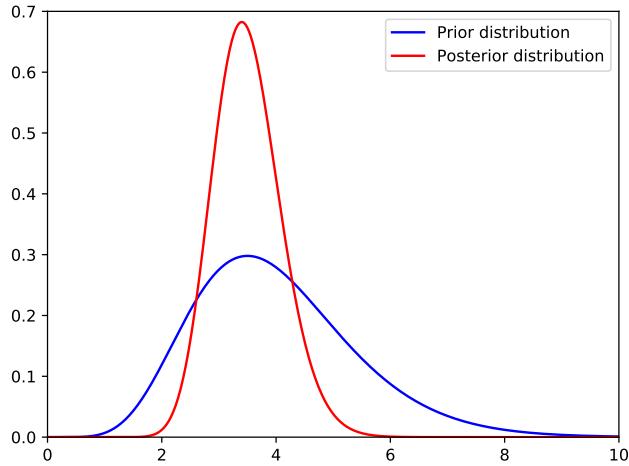


Figure 5.16: The prior distribution in blue and the posterior in red for Bayesian estimation of a Poisson distribution gamma conjugacy.

```

20
21 newAlpha, newBeta = alpha + sum(data), beta + length(data)
22 closedFormBayesEstimate = mean(Gamma(newAlpha, 1/newBeta))
23
24 println("Computational Bayes Estimate: ", bayesEstimate)
25 println("Closed form Bayes Estimate: ", closedFormBayesEstimate)

```

```

Computational Bayes Estimate: 3.49999999999874
Closed form Bayes Estimate: 3.5

```

- In lines 3 and 4 the hyperparameters and prior are defined.
- In lines 7 to 17 the estimate is calculated in the brute force same manner as listing 5.18:
- In line 19 the hyperparameters are updated using the rule for the case of Gamma-poisson.
- In line 21 the mean (Bayes estimate) is computed, this time symbolically using the `mean()` function applied to a gamma distribution.
- The value calculated from both the brute force method and via the simple update rule are printed in lines 23 and 24. The results illustrate the validity of the conjugacy based update of the hyperparamters.

Monte Carlo Markov Chains

In many applicative cases of Bayesian statistics, sterile situations of conjugate priors are not available, yet computation of posterior distributions and Bayes estimates are needed. In cases where the dimension of the parameter space is high, carrying out straightforward integration as done in

Listing 5.18 is not possible. However, there are other ways of carrying out Bayesian inference. One such popular way is by using algorithms that fall under the category known as *Monte Carlo Markov Chains*, MCMC, (also known as *Markov Chain Monte Carlo* with a different word order).

The *Metropolis Hastings* algorithm is one such popular MCMC algorithm. It produces a series of samples $\theta(1), \theta(2), \theta(3), \dots$ where it is guaranteed that for large t , $\theta(t)$ is distributed according to the posterior distribution. Technically, the random sequence $\{\theta(t)\}_{t=1}^{\infty}$ is a Markov chain (see Chapter 9 for more details about Markov chains) and it is guaranteed that the stationary distribution of this Markov chain is the posterior distribution. Hence the posterior distribution is an input parameter to the algorithm. The big virtue is that the algorithm only uses ratios of posterior on different parameter values. For example, for parameter values θ_1 and θ_2 , the algorithm only uses the posterior distribution via the ratio,

$$L(\theta_1, \theta_2) = \frac{f(\theta_1 | x)}{f(\theta_2 | x)}.$$

This means that the normalizing constant is not needed as it is implicitly cancelled out. Thus using the posterior in the form (5.20) suffices.

Further to the posterior distribution, an additional input parameter to Metropolis Hastings is the so called *proposal density*, denoted by $q(\cdot | \cdot)$. This is a family of probability distributions where given a certain value of θ_1 taken as a parameter, the new value, say θ_2 , is distributed with PDF,

$$q(\theta_2 | \theta_1).$$

The idea of Metropolis Hastings is to walk around the parameter space by randomly generating new values using $q(\cdot | \cdot)$. Some new values are accepted while others are not, all with a manner which ensures the desired limiting behavior. Acceptance or rejection is carried out with probability,

$$H = \min \left\{ 1, L(\theta^*, \theta(t)) \frac{q(\theta(t) | \theta^*)}{q(\theta^* | \theta(t))} \right\},$$

where θ^* is the new proposed value, generated via $q(\cdot | \theta(t))$, and $\theta(t)$ is the current value. With each such iteration, the new value is accepted with probability H and otherwise rejected. With certain technical properties of the posterior and proposal densities, the theory of Markov chains then guarantees that the stationary distribution of the sequence $\{\theta(t)\}$ is the posterior distribution.

Different variants of the Metropolis Hastings algorithm employ different types of proposal densities. There are also generalizations and extensions that we don't discuss here, such as *Gibbs Sampling* and *Hamiltonian Monte Carlo* for example.

To help illustrate some of these concepts, we now implement a simple version of Metropolis Hastings where we use the *folded normal distribution* as a proposal density. This distribution is achieved by taking a normal random variable X with mean μ and variance σ^2 and considering $Y = |X|$. In this case, the PDF of Y is,

$$f(y) = \frac{1}{\sigma\sqrt{2\pi}} \left(e^{-\frac{(x-\mu)^2}{2\sigma^2}} + e^{-\frac{(x+\mu)^2}{2\sigma^2}} \right). \quad (5.23)$$

Our choice of this specific density is purely for simplicity of implementation, and in addition it suits the case that we demonstrate, where the support of the parameter in question is non-negative.

In Listing 5.20 we implement the Metropolis Hastings algorithm, and show that we obtain the same numerical results as we did using gamma conjugacy. The histogram of the samples is plotted in Figure 5.17.

Listing 5.20: Bayesian inference using MCMC

```

1  using Distributions, PyPlot
2
3  alpha, beta = 8, 2
4  prior(lam) = pdf(Gamma(alpha, 1/beta), lam)
5  data = [3, 7, 0, 1, 5, 3, 6, 2]
6
7  likelihood(lam) = *[pdf(Poisson(lam),x) for x in data]...
8  posteriorUpToK(lam) = likelihood(lam)*prior(lam)
9
10 sig = 0.5
11 foldedNormalPDF(x,mu) = (1/sqrt(2*pi*sig^2))*(exp(-(x-mu)^2/2sig^2)
12                                + exp(-(x+mu)^2/2sig^2))
13 foldedNormalRV(mu) = abs(rand(Normal(mu,sig)))
14
15 function sampler(piProb,qProp,rvProp)
16     lam = 1
17     warmN, N = 10^5, 10^6
18     samples = zeros(N-warmN)
19
20     for t in 1:N
21         while true
22             lamTry = rvProp(lam)
23             L = piProb(lamTry)/piProb(lam)
24             H = min(1,L*qProp(lam, lamTry)/qProp(lamTry, lam))
25             if rand() < H
26                 lam = lamTry
27                 if t > warmN
28                     samples[t-warmN] = lam
29                 end
30                 break
31             end
32         end
33     end
34     return samples
35 end
36
37 mcmcSamples = sampler(posteriorUpToK,foldedNormalPDF,foldedNormalRV)
38 plt[:hist](mcmcSamples,100,density=true, label="Histogram of MCMC samples");
39
40 lamRange = 0:0.01:10
41 plot(lamRange, prior.(lamRange), "b", label="Prior distribution")
42 closedFormPosterior(lam)=pdf(Gamma(alpha + sum(data),1/(beta+length(data))),lam)
43 plot(lamRange, closedFormPosterior.(lamRange), "r", label="Posterior distribution")
44 xlim(0, 10); ylim(0, 0.7); legend(loc="upper right")
45 println("MCMC Bayes Estimate: ",mean(mcmcSamples))
```

MCMC Bayes Estimate: 3.5093710080931633

- Lines 3-8 are similar to the previous listings 5.18 and 5.19.

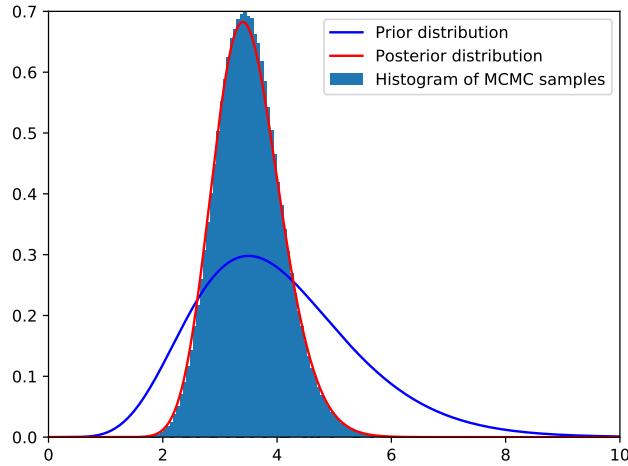


Figure 5.17: The prior distribution in blue and the posterior in red. Monte Carlo Markov Chain samples are generated using Metropolis Hastings and their histogram plotted in blue.

- In lines 10-13 the proposal density `foldedNormalPDF()` is defined in accordance with (5.23), along with a function for generating a proposal random variable, `foldedNormalRV()`.
- Lines 15-35 define the function `sampler()`. It operates on a desired (non-normalized) density, `piProb` and runs the Metropolis Hastings algorithm for sampling from that density. The argument, `qProp`, is the proposal density and the argument `rvProp` is for generating from the proposal. All three arguments are assumed to be functions which `sampler()` invokes.
- Our implementation uses a *warm up sequence* with a length specified by `warmN` in line 17. The idea here is to let the algorithm run for a while to remove any bias introduced by initial values.
- Lines 20-33 constitute the main loop over N samples generated by the algorithm. In our implementation, we setup an internal loop (lines 21-32) that iterates until a proposal is accepted (and breaks in line 30).
- Line 45 prints the Bayes estimate. As can be seen, it agrees with the estimate of the previous example.

Chapter 6

Confidence Intervals - DRAFT

We now visit a variety of confidence intervals used in standard statistical procedures. As introduced in Section 5.5, a confidence interval with a confidence level $1 - \alpha$ is an interval $[L, U]$ resulting from the observations. When considering confidence intervals in the setting of symmetric sampling distributions (as is the case for most of this chapter), a typical formula for $[L, U]$ is of the form,

$$\hat{\theta} \pm K_\alpha \text{serr}. \quad (6.1)$$

Here $\hat{\theta}$ is typically the point estimate for the parameter at hand, serr is some measure or estimate of the variability (e.g. standard error), and K_α is a constant depending on the model at hand and on α . Typically by decreasing $\alpha \rightarrow 0$, we have that K_α increases, implying a wider confidence interval. For the examples in this chapter, typical values for K_α are in the range of [1.5, 3.5] for values of α in the range of [0.01, 0.1]. Most of the confidence intervals presented in this chapter follow the form (6.1), with the specific form of K_α often depending on assumptions such as:

Sample size: Small samples vs. large samples.

Variance: Variance, σ^2 known, versus unknown.

Distribution: Normally distributed data vs. not.

To explore confidence intervals with Julia, we compute both the confidence intervals using standard statistical formulas, and illustrate how they can be obtained using the `HypothesisTests` package. As we shall see, this package includes various functions that generate objects resulting from a statistical procedure. We can either look at the output of these objects, or query them using other functions, specifically the `confint()` function.

The individual sections of this chapter focus on specific confidence intervals and general concepts. In Section 6.1 we cover confidence intervals for the mean of a single population. In Section 6.2 we deal with comparisons of means of two populations. In Section 6.3 we present the bootstrap method, a general methodology for creating confidence intervals. In Section 6.4 we gain a better understanding of model assumptions via the example of a confidence interval for the variance. In Section 6.5 we present *prediction intervals*, a concept dealing with prediction of future observations based on previous ones. We close with Section 6.6, coming from Bayesian statistics.

6.1 Single Sample Confidence Intervals for the Mean

Let us first consider the case where we wish to estimate the mean, μ from a random sample, X_1, \dots, X_n . As covered previously, a point estimate for the mean is the sample mean \bar{X} . A typical formula for the confidence interval of the mean is then,

$$\bar{X} \pm K_\alpha \frac{S}{\sqrt{n}}. \quad (6.2)$$

Here the bounds around the point estimator \bar{X} are defined by the addition and subtraction of a multiple, K_α , of the standard error, S/\sqrt{n} (first introduced in Section 4.2). The multiple K_α takes on different forms depending on the specific case at hand.

Population Variance Known

If we assume that the population variance, σ^2 , is known and the data is normally distributed, then the sample mean, \bar{X} , is normally distributed with mean μ and variance σ^2/n . This yields,

$$\mathbb{P}\left(\mu - z_{1-\frac{\alpha}{2}} \frac{\sigma}{\sqrt{n}} \leq \bar{X} \leq \mu + z_{1-\frac{\alpha}{2}} \frac{\sigma}{\sqrt{n}}\right) = 1 - \alpha, \quad (6.3)$$

where $z_{1-\frac{\alpha}{2}}$ is the $1 - \frac{\alpha}{2}$ quantile of the standard normal distribution. In Julia this is computed via `quantile(1-alpha/2)`. Denote now actual sample mean estimate obtained from data by \bar{x} . Then, by rearranging the inequalities inside the probability statement above, we obtain the following confidence interval formula,

$$\bar{x} \pm z_{1-\frac{\alpha}{2}} \frac{\sigma}{\sqrt{n}}. \quad (6.4)$$

In practice, σ^2 is rarely known, hence it is tempting to replace σ by s (sample standard deviation), in the formula above. Such a replacement is generally fine for large samples. In the case of small samples, one should use the case of population variance unknown, covered at the end of this section.

Also, consider the normality assumption. In cases where the data is not normally distributed, the probability statement (6.3) only approximately holds. However, as $n \rightarrow \infty$, it quickly becomes precise due to the central limit theorem. Hence, the confidence interval (6.4) may be used for non-small samples.

In Julia, computation of confidence intervals is done using functions from the `HypothesisTests` package (even when we don't carry out an hypothesis test). The code in Listing 6.1 below illustrates computation of the confidence interval (6.4) using both the package and evaluating the formula directly. It can be observed that the direct computation and the use of the `confint()` function yield the same result.

Listing 6.1: CI for single sample population with variance assumed known

```

1  using CSV, Distributions, HypothesisTests
2
3  data = CSV.read("machine1.csv", header=false)[:,1]
4  xBar, n = mean(data), length(data)
5  sig = 1.2
6  alpha = 0.1

```

```

7 z = quantile(Normal(),1-alpha/2)
8
9 println("Calculating formula: ", (xBar - z*sig/sqrt(n), xBar + z*sig/sqrt(n)))
10 println("Using confint() function: ", confint(OneSampleZTest(xBar,sig,n),alpha))

```

```

Calculating formula:      (52.51484557853184, 53.397566664027984)
Using confint() function: (52.51484557853184, 53.397566664027984)

```

- In line 1 we call the required packages, including the CSV package.
- In line 3 we load in our data. Note the use of the header=false argument, and also the trailing [:, 1] which is used to select all rows of the data.
- In line 4 we calculate the sample mean, and the number of observations in our sample.
- In lines 5, we stipulate the standard deviation as 1.2, as this scenario is one in which the population standard deviation, or population variance, is assumed known.
- In line 7 we calculate the value of z for $1 - \alpha/2$. This isn't a quantity that depends on the sample, but rather is a fixed number. Indeed as is well known from statistical tables it equals approximately 1.65 when $\alpha = 10\%$.
- In line 9 the formula for the confidence interval (6.4) is evaluated directly.
- In line 10 the function OneSampleZTest() is first used to conduct a one sample z-test given the parameters xBar, sig, and n. The confint() function is then applied to this output, for the specified value of alpha. It can be observed that the two methods are in agreement. Note that hypothesis tests are covered further in Chapter 7)

Population Variance Unknown

A celebrated procedure in elementary statistics is the T-distribution based confidence interval. Here we relax the assumptions of the previous confidence interval allowing σ^2 to be an unknown quantity. In such a case, if we replace σ by the sample standard deviation s , then the probability statement (6.3) no longer holds. However, by using the T-distribution (see Section 5.2) we are able to correct the confidence interval to,

$$\bar{x} \pm t_{1-\frac{\alpha}{2},n-1} \frac{s}{\sqrt{n}}. \quad (6.5)$$

Here, $t_{1-\frac{\alpha}{2},n-1}$ is the $1 - \frac{\alpha}{2}$ quantile of a T-distribution with $n - 1$ degrees of freedom. This can be expressed in Julia as `quantile(TDist(n-1),1-alpha/2)`.

For small samples, the replacement of $z_{1-\frac{\alpha}{2}}$ by $t_{1-\frac{\alpha}{2},n-1}$ significantly affects the width of the confidence interval, as for the same value of α , the T case is wider. However, as $n \rightarrow \infty$, we have, $t_{1-\frac{\alpha}{2},n-1} \rightarrow z_{1-\frac{\alpha}{2}}$, as illustrated in Figure 5.6. Hence for non-small samples, the confidence interval (6.5) is very close to the confidence interval (6.4) with s replacing σ . Note that the t-confidence interval hinges on the normality assumption of the data. In fact for small samples, cases that deviates from normality imply imprecision of the confidence intervals. However for larger samples,

these confidence intervals serve as a good approximation. However for larger samples one might as well use $z_{1-\frac{\alpha}{2}}$ instead of $t_{1-\frac{\alpha}{2},n-1}$.

The code in Listing 6.1 calculates the confidence interval (6.5), where it is assumed that the population variance is unknown.

Listing 6.2: CI for single sample population with variance assumed unknown

```

1  using CSV, Distributions, HypothesisTests
2
3  data = CSV.read("machine1.csv", header=false)[:,1]
4  xBar, n = mean(data), length(data)
5  s = std(data)
6  alpha = 0.1
7  t = quantile(TDist(n-1), 1-alpha/2)
8
9  println("Calculating formula: ", (xBar - t*s/sqrt(n), xBar + t*s/sqrt(n)))
10 println("Using confint() function: ", confint(OneSampleTTest(xBar, s, n), alpha))

```

```

Calculating formula: (52.49989385779555, 53.412518384764276)
Using confint() function: (52.49989385779555, 53.412518384764276)

```

- This example is very similar to Listing 6.1, however there are several differences.
- In line 5, since the population variance is assumed unknown, the population standard deviation `sig` is replaced with the sample standard deviation `s`.
- In addition, line 7 the quantile `t` is calculated on a T-distribution, `TDist(n-1)`, with $n-1$ degrees of freedom. Previously, the quantile `z` was calculated on a standard normal distribution `Normal()`.
- Lines 9 and 10 are very similar to those in the previous listing, but `z` and `sig` are replaced with `t` and `s` respectively. It can be seen that the outputs of lines 9 and 10 are in agreement, and that the confidence interval is wider than that calculated in the previous Listing 6.1.

6.2 Two Sample Confidence Intervals for the Difference in Means

We now consider cases in which there are two populations involved. As an example, consider two separate machines, 1 and 2, which are designed to make pipes of the same diameter. In this case, due to small differences and tolerances in the manufacturing process, the distribution of pipe diameters from each machine will differ. In such cases where two populations are involved, it is often of interest to estimate the difference between the population means, $\mu_1 - \mu_2$.

In order to do this we first collect two random samples, $x_{1,1}, \dots, x_{n_1,1}$ and $x_{1,2}, \dots, x_{n_2,2}$. For each sample $i = 1, 2$ we have the sample mean \bar{x}_i , and sample standard deviation s_i . In addition, the difference in sample means, $\bar{x}_1 - \bar{x}_2$ serves as a point estimate for the difference in population means, $\mu_1 - \mu_2$.

A confidence interval around this point is then constructed via the same process seen previously,

$$\bar{x}_1 - \bar{x}_2 \pm K_\alpha s_{\text{err}}.$$

We now elaborate on the values of K_α and s_{err} based on model assumptions.

Population Variances Known

In the (unrealistic) case that the population variances are known, we may explicitly compute,

$$\text{Var}(\bar{X}_1 - \bar{X}_2) = \frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}.$$

Hence the standard error is given by,

$$s_{\text{err}} = \sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}. \quad (6.6)$$

When combined with the assumption that the data is normally distributed, we can derive the following confidence interval,

$$\bar{x}_1 - \bar{x}_2 \pm z_{1-\frac{\alpha}{2}} \sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}. \quad (6.7)$$

While this case is often not applicable in practice, it is useful to cover for pedagogical reasons. Due to the fact that the population variances are almost always unknown, the `HypothesisTests` package in Julia does not have a function for this case. However, for completeness, we evaluate equation (6.7) manually in Listing 6.3 below.

Listing 6.3: CI for difference in population means with variance known

```

1  using CSV, Distributions, HypothesisTests
2
3  data1 = CSV.read("machine1.csv", header=false)[:,1]
4  data2 = CSV.read("machine2.csv", header=false)[:,1]
5  xBar1, xBar2 = mean(data1), mean(data2)
6  n1, n2 = length(data1), length(data2)
7  sig1, sig2 = 1.2, 1.6
8  alpha = 0.05
9  z = quantile(Normal(), 1-alpha/2)
10
11 println("Calculating formula: ", (xBar1 - xBar2 - z*sqrt(sig1^2/n1+sig2^2/n2),
12                                     xBar1 - xBar2 + z*sqrt(sig1^2/n1+sig2^2/n2)))

```

Calculating formula: (1.1016568035908845, 2.9159620096069574)

- This listing is similar to those previously covered in this chapter.
- The data for our two samples is first loaded in lines 3-4 (note the use of the `header=false` argument, so that an array of floats is returned).
- The sample means and number of observations calculated in lines 5-6.

- In line 7, we stipulate the standard deviations of both populations 1 and 2, as 1.2 and 1.6 respectively (since this scenario is one in which the population standard deviation, or population variance, is assumed known).
- In line 10 equation (6.7) is evaluated manually.

Population Variances Unknown and Assumed Equal

Typically, when considering cases consisting of two populations, the population variances are unknown. In such cases, a common and practical assumption is that the variances are equal, denoted by σ^2 . Based on this assumption, it is sensible to use both sample variances in the estimate of σ^2 . This estimated variance of both samples is known as the *pooled sample variance*, and is given by,

$$S_p^2 = \frac{(n_1 - 1)S_1^2 + (n_2 - 1)S_2^2}{n_1 + n_2 - 2}.$$

Upon closer inspection, it can be observed that the above is in fact a weighted average of the sample variances of the individual samples.

In this case, it can be shown that,

$$T = \frac{\bar{X}_1 - \bar{X}_2 - (\mu_1 - \mu_2)}{S_{\text{err}}} \quad (6.8)$$

is distributed according to a T-distribution with $n_1 + n_2 - 2$ degrees of freedom. The standard error is then,

$$S_{\text{err}} = S_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}.$$

Hence we arrive at the following confidence interval,

$$\bar{x}_1 - \bar{x}_2 \pm t_{1-\frac{\alpha}{2}, n_1+n_2-2} s_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}. \quad (6.9)$$

The code in Listing 6.4 below calculates the confidence interval (6.9), where it is assumed that the population variance is unknown, and compares this with those resulting from the use of the HypothesisTests package.

Listing 6.4: CI for difference in means with variance unknown and assumed equal

```

1  using CSV, Distributions, HypothesisTests
2
3  data1 = CSV.read("machine1.csv", header=false, allowmissing=:none)[:,1]
4  data2 = CSV.read("machine2.csv", header=false, allowmissing=:none)[:,1]
5  xBar1, xBar2 = mean(data1), mean(data2)
6  n1, n2 = length(data1), length(data2)
7  alpha = 0.05
8  t = quantile(TDist(n1+n2-2), 1-alpha/2)
9
10 s1, s2 = std(data1), std(data2)
11 sP = sqrt(((n1-1)*s1^2 + (n2-1)*s2^2) / (n1+n2-2))
12

```

```

13  println("Calculating formula: ", (xBar1 - xBar2 - t*sP* sqrt(1/n1 + 1/n2),
14                      xBar1 - xBar2 + t*sP* sqrt(1/n1 + 1/n2)))
15  println("Using confint(): ", confint(EqualVarianceTTest(data1,data2),alpha))

```

```

Calculating formula:      (1.1127539574575822, 2.90486485574026)
Using confint() function: (1.1127539574575822, 2.90486485574026)

```

- In line 8, a T-distribution with n_1+n_2-2 degrees of freedom is used.
- In line 10 the sample variances are calculated.
- In line 11, the pooled sample variance s_P is calculated.
- In lines 13-14, equation (6.9) is evaluated manually, while in line 15 the `confint()` function is used. It can be observed that the results are in agreement.

Population Variances Unknown and not Assumed Equal

In certain cases, it may be appropriate to relax the assumption of equal population variances. In this case, the estimate for S_{err} is given by,

$$S_{\text{err}} = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}.$$

This is due to the fact that the variance of the difference of two independent sample means is the sum of the variances. Hence in this case, the statistic (6.8), written as,

$$T = \frac{\bar{X}_1 - \bar{X}_2 - (\mu_1 - \mu_2)}{\sqrt{\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}}}, \quad (6.10)$$

is only t-distributed in the case of variances equal - otherwise it isn't. Nevertheless, an approximate confidence interval is commonly used by approximating the distribution of (6.10) with a T-distribution. This is called the *Satterthwaite approximation*.

The approximation suggests a T-distribution with a parameter (degrees of freedom) given via,

$$v = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{\left(s_1^2/n_1\right)^2}{n_1-1} + \frac{\left(s_2^2/n_2\right)^2}{n_2-1}}. \quad (6.11)$$

Now it holds that,

$$T \underset{\text{approx}}{\sim} t(v). \quad (6.12)$$

That is, the random variable T from equation (6.10) is approximately distributed according to a t -distribution with v degrees of freedom (note that v does not need to be an integer). We investigate this approximation further in Listing 6.6 later.

Given (6.12), we arrive at the following confidence interval,

$$\bar{x}_1 - \bar{x}_2 \pm t_{1-\frac{\alpha}{2}, v} \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}. \quad (6.13)$$

In Listing 6.5 below, the confidence interval (6.13), where it is assumed that the population variance is unknown, and compares the result to those resulting from the use of functions from HypothesisTests.

Listing 6.5: CI for difference in means with variance unknown and not assumed equal

```

1  using CSV, Distributions, HypothesisTests
2
3  data1 = CSV.read("machine1.csv", header=false, allowmissing=:none)[:,1]
4  data2 = CSV.read("machine2.csv", header=false, allowmissing=:none)[:,1]
5  xBar1, xBar2 = mean(data1), mean(data2)
6  s1, s2 = std(data1), std(data2)
7  n1, n2 = length(data1), length(data2)
8  alpha = 0.05
9
10 v = (s1^2/n1 + s2^2/n2)^2 / ((s1^2/n1)^2 / (n1-1) + (s2^2/n2)^2 / (n2-1))
11
12 t = quantile(TDist(v), 1-alpha/2)
13
14 println("Calculating formula: ", (xBar1 - xBar2 - t*sqrt(s1^2/n1 + s2^2/n2),
15                                xBar1 - xBar2 + t*sqrt(s1^2/n1 + s2^2/n2)))
16 println("Using confint(): ", confint(UnequalVarianceTTest(data1, data2), alpha))

```

```

Calculating formula: (1.0960161148824918, 2.9216026983153505)
Using confint():      (1.0960161148824918, 2.9216026983153505)

```

- The main difference in is code block is the calculation of the degrees of freedom, v , which is performed in line 10.
- In line 12 v is then used to derive the T-statistic t .
- In lines 14-15, equation (6.13) is evaluated manually, while in line 16 the `confint()` function is used. It can be observed that the results are in agreement.

The Validity of the Satterthwaite Approximation

We now investigate the approximate distribution of (6.12). Observe that both sides of the “distributed as” (\sim) symbol are random variables which depend on the same random experiment. That is, the statement can be presented generally, as a case of the following format,

$$X(\omega) \sim F_{h(\omega)}, \quad (6.14)$$

where ω is a point in the sample space (see Chapter 2). Here $X(\omega)$ is a random variable, and F is a distribution that depends on a parameter h , which itself depends on ω . In our case, h is given

by (6.11). That is, in our example, h can be thought of as v , which itself depends on the specific observations made for our two sample groups (a function of s_1 and s_2).

Now by recalling the inverse probability transform from Section 3.4, we have that (6.14) is equivalent to,

$$F_{h(\omega)}^{-1}(X(\omega)) \sim \text{Uniform}(0, 1). \quad (6.15)$$

We thus expect that (6.15) hold approximately. This is different from the naive alternative of treating h as simply dependent on the number of observations made (n_1 and n_2), as in this case, the distribution is not expected to be uniform.

We investigate this in Listing 6.6 below, where we construct a QQ plot of T-values calculated from the Satterthwaite approximation (6.12) and compare this with those calculated the naive equal variance case. We observe that those calculated via Satterthwaite approximation are a better approximation.

Listing 6.6: Analyzing the Satterthwaite approximation

```

1  using Distributions, PyPlot
2
3  mu1, sig1, n1 = 0, 2, 8
4  mu2, sig2, n2 = 0, 30, 15
5  dist1 = Normal(mu1, sig1)
6  dist2 = Normal(mu2, sig2)
7
8  N = 10^6
9  tdArray = Array{Tuple{Float64,Float64}}(undef,N)
10
11 df(s1,s2,n1,n2) =
12     (s1^2/n1 + s2^2/n2)^2 / ( (s1^2/n1)^2/(n1-1) + (s2^2/n2)^2/(n2-1) )
13
14 for i in 1:N
15     x1Data = rand(dist1, n1)
16     x2Data = rand(dist2, n2)
17
18     x1Bar, x2Bar = mean(x1Data), mean(x2Data)
19     s1, s2 = std(x1Data), std(x2Data)
20
21     tStat = (x1Bar - x2Bar) / sqrt(s1^2/n1 + s2^2/n2)
22
23     tdArray[i] = (tStat , df(s1,s2,n1,n2))
24 end
25 sort!(tdArray, by = tdArray -> tdArray[1])
26
27 invVal(data,i) = quantile(TDist(data),i/(N+1))
28
29 xCoords = Array{Float64}(undef,N)
30 yCoords1 = Array{Float64}(undef,N)
31 yCoords2 = Array{Float64}(undef,N)
32
33 for i in 1:N
34     xCoords[i] = first(tdArray[i])
35     yCoords1[i] = invVal(last(tdArray[i]),i)
36     yCoords2[i] = invVal(n1+n2-2,i)
37 end
38
39 plot(xCoords,yCoords1,label="Calculated v", "b.", ms="1.5")

```

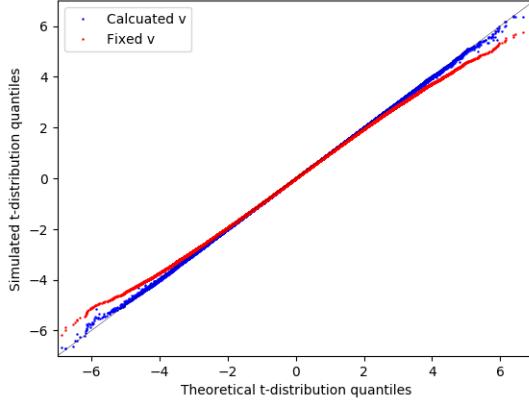


Figure 6.1: Comparing QQ plots of T-statistics from identical experiments, given Satterthwaite calculated v 's, along with T-statistics given constant v . It can be observed that the distribution of T-statistics from Satterthwaite calculated v 's yields better results than the constant v case.

```

40 plot(xCoords,yCoords2,label="Fixed v", "r.",ms="1.5")
41 plot([-10,10], [-10,10], "k",lw="0.3")
42 legend(loc="upper left")
43 xlim(-7,7)
44 ylim(-7,7)
45 xlabel("Theoretical t-distribution quantiles")
46 ylabel("Simulated t-distribution quantiles")

```

- In lines 3-4 we set the means and standard deviations of the two underlying processes from which we will be making observations. We also specify the number of observations that will be made for each group.
- In line 8-9 we set the number of times we repeat the experiment, N , and pre-allocate the array $tdArray$, in which each element is a pair of tuples. The first element will be the T-statistic calculated via (6.10), while the second element will be the corresponding degrees of freedom calculated via (6.11).
- In line 11 we define the function $df()$, which calculates the degrees of freedom via (6.11).
- In lines 14-24, we conduct N experiments, where for each we calculate the T-statistic, and the degrees of freedom. The T-statistic is calculated on line 21, while the degrees of freedom are calculated via the $df()$ function on line 23.
- In line 25 the function $sort!()$ is used to re-order the array $tdArrray$ in ascending order according to the T-statistics. This is done so that later we can construct the QQ plot.
- In line 27 the function $invVal()$ is defined, which uses the $quantile()$ function to perform the inverse probability transform on the degrees of freedom associated with each T-statistic for each experiment. Note that the number of quantiles is one more than the number of experiments, i.e. $N+1$.
- In lines 33-37 the quantiles of our data are calculated. The array $xCoords$ represent the T-statistic quantiles, while the array $yCoords1$ represent the quantiles of a T-distribution with

v degrees of freedom, where v is calculated via (6.11). The array `yCoords2` on the other hand represent the quantiles of a T-distribution with v degrees of freedom, where $v = n_1 + n_2 - 2$.

- It can be observed that the data from the fixed v case deviates further from the 1:1 slope than that where v was calculated based on each experiments sample observations (i.e. calculated from equation (6.11)). This indicates that the Satterthwaite approximation is a better approximation than simply using the degrees of freedom.

6.3 Bootstrap Confidence Intervals

As is typical when developing confidence intervals, the main goal is to make some sort of inference about the population based on sample data. However, when a statistical model is not readily available and/or the confidence intervals are algebraically complex we seek alternative methods. One such general method is the method of *bootstrap confidence intervals*.

Bootstrap is a useful technique, which relies on resampling from the sample data, in order to make inferences about the population. There are several ways to resample the data. These include applying the inverse probability transform, or generating multiple groups of sample observations by uniformly sampling from the sample data set. The bootstrap confidence interval is defined as the lower and upper $(\frac{\alpha}{2}, 1 - \frac{\alpha}{2})$ quantiles respectively of the sampling distribution of the estimator in question.

In Listing 6.7 below, we generate a bootstrapped confidence interval for the mean of the data that was used in Section 6.1. By resampling the sample data, and calculating N bootstrapped sample means, we are able to generate an approximate sampling distribution of bootstrapped sample means. We then use the `quantile()` function to calculate the lower and upper bounds of our confidence interval, given a specific value of α .

Listing 6.7: Bootstrap confidence interval

```

1  using Random, CSV, Distributions, PyPlot
2  Random.seed!(0)
3
4  sampleData = CSV.read("machine1.csv", header=false)[:,1]
5  n, N = length(sampleData), 10^6
6  alpha = 0.1
7
8  bootstrapSampleMeans = [mean(rand(sampleData, n)) for i in 1:N]
9  L = quantile(bootstrapSampleMeans, alpha/2)
10 U = quantile(bootstrapSampleMeans, 1-alpha/2)
11
12 plt[:hist](bootstrapSampleMeans, 1000, color="blue",
13             normed=true, histtype = "step", label="Sample \nmeans")
14 plot([L, L], [0,2], "k--", label="95% CI")
15 plot([U, U], [0,2], "k--")
16 xlim(52,54)
17 ylim(0,2)
18 legend(loc="upper right");

```

- In line 4 we load our sample observations, and store them in the variable `sampleData`. In

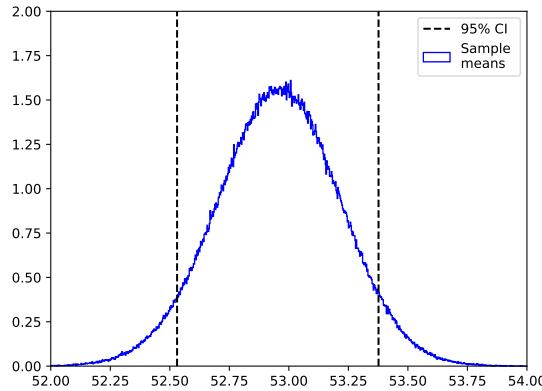


Figure 6.2: A single CI generated by bootstrapped data.

line 5, the total number of sample observations is assigned to n .

- In line 6 we specify the level of our confidence interval α .
- In line 8 we generate N bootstrapped sample means. By uniformly and randomly sampling n observations from our `sampleData`, and calculating the mean each time, we essentially perform inverse transform sampling. The bootstrapped sample means are assigned as the array `bootstrapSampleMeans`.
- In lines 9-10 the lower and upper quantiles of our bootstrapped sample data is calculated, and stored as the variables `L` and `U` respectively.
- In line 12-15 a histogram of `bootstrapSampleMeans` is plotted, along with the lower and upper quantiles `L` and `U` calculated.

One may notice that in line 8 of Listing 6.7 above, that the bootstrapped confidence interval is dependent on the amount of sample observations available, n . If the number of sample observations is not very large, then the *coverage probability* of bootstrapped confidence interval is only approximately at $1 - \alpha$, but not exactly at that value.

In Listing 6.8 below, we investigate the sensitivity of the number of observations on the coverage probability. We create a series of confidence intervals based on different numbers of sample observations from an exponential distribution with a mean of 10. We then compare the proportion of confidence intervals which cover our parameter, and observe that as the number of sample observations increases, the coverage probability approaches $1 - \alpha$.

Listing 6.8: Coverage probability for bootstrap confidence intervals

```

1  using Random, Distributions
2  Random.seed!(0)
3
4  M = 1000
5  N = 10^4
6  nRange = 5:5:50
7  alpha = 0.1
8

```

```

9   for n in nRange
10    coverageCount = 0
11    for i in 1:M
12      sampleData = rand(Exponential(10), n)
13      bootstrapSampleMeans = [mean(rand(sampleData, n)) for _ in 1:N]
14      L = quantile(bootstrapSampleMeans, alpha/2)
15      U = quantile(bootstrapSampleMeans, 1-alpha/2)
16      coverageCount += L < 10 && 10 < U
17    end
18    println("n = ", n, "\t coverage = ", coverageCount/M)
19 end

```

```

n = 5      coverage = 0.771
n = 10     coverage = 0.81
n = 15     coverage = 0.836
n = 20     coverage = 0.84
n = 25     coverage = 0.859
n = 30     coverage = 0.879
n = 35     coverage = 0.867
n = 40     coverage = 0.878
n = 45     coverage = 0.863
n = 50     coverage = 0.872

```

- In line 4 we specify the number of bootstrapped confidence intervals we will generate, M, for each case of n observations.
- In line 5, we specify that each confidence interval will be based on N bootstrapped sample means.
- In line 6 we specify the range of values of n sample observations that we will consider, from 5 to 50, in increments of 5.
- In line 7 we specify an alpha value of 0.1
- In lines 9-19, we cycle through each value of n observations in nRange, and for each case, count the number confidence intervals which contain the parameter in question (i.e. the mean).
- In line 10 we set the counter coverageCount to zero. This will be incremented by one each time a confidence interval covers our parameter in question (i.e 10).
- In lines 11-17, we generate M bootstrapped confidence intervals and count the proportion of times our parameter (mean of 10) are contained within them. For each case, we first generate n sample observations from our exponential distribution, stored as sampleData. This data is then used to generate N bootstrapped sample means, stored as bootstrapSampleMeans. The lower and upper quantiles of bootstrapSampleMeans is then calculated, and if our parameter (mean of 10) is contained within these bounds, coverageCount is incremented by 1.

6.4 Confidence Interval for the Variance of Normal Population

Consider sampling from a population that follows a normal distribution. A point estimator for the population variance is the sample variance,

$$S^2 = \frac{1}{(n-1)} \sum_{i=1}^n (X_i - \bar{X})^2.$$

As shown in 5.2,

$$\frac{(n-1)S^2}{\sigma^2} \sim \chi_{n-1}^2.$$

Therefore

$$\mathbb{P}\left(\chi_{\frac{\alpha}{2}, n-1}^2 < \frac{(n-1)S^2}{\sigma^2} < \chi_{1-\frac{\alpha}{2}, n-1}^2\right) = 1 - \alpha. \quad (6.16)$$

Hence we can re-arrange to obtain a two-sided $100(1 - \alpha)\%$ confidence interval for the variance of a normal population (denoting by s^2 the observed estimator),

$$\frac{(n-1)s^2}{\chi_{1-\frac{\alpha}{2}, n-1}^2} < \sigma^2 < \frac{(n-1)s^2}{\chi_{\frac{\alpha}{2}, n-1}^2}. \quad (6.17)$$

Note that (6.16) only holds when sampling from data that is normally distributed. If the data is not normally distributed, then our confidence intervals will be inaccurate. This concept is explored further below.

Sensitivity of the Normality Assumption

We now look at the sensitivity of the normality assumption on the confidence interval for the variance. As part of this example we first introduce the *logistic distribution*. This distribution has a “bell curved” shape and is defined by the two parameters, μ and s , the location and scale parameters respectively. The PDF of the logistic distribution is,

$$f(x) = \frac{e^{-\frac{x-\mu}{s}}}{s \left(1 + e^{-\frac{x-\mu}{s}}\right)^2}, \quad (6.18)$$

with the variance given by $s^2\pi^2/3$.

In Listing 6.9 below, the PDF of a normal distribution with mean 2 and variance 3^2 is plotted against that of a logistic distribution with the same mean and variance. While both distributions share the same mean and variance, their sample variances are significantly different. To see this we consider a sample size of $n = 15$ and plot histograms of the resulting sample variances.

Listing 6.9: Comparison of sample variance distributions

```
1  using Distributions, PyPlot
```

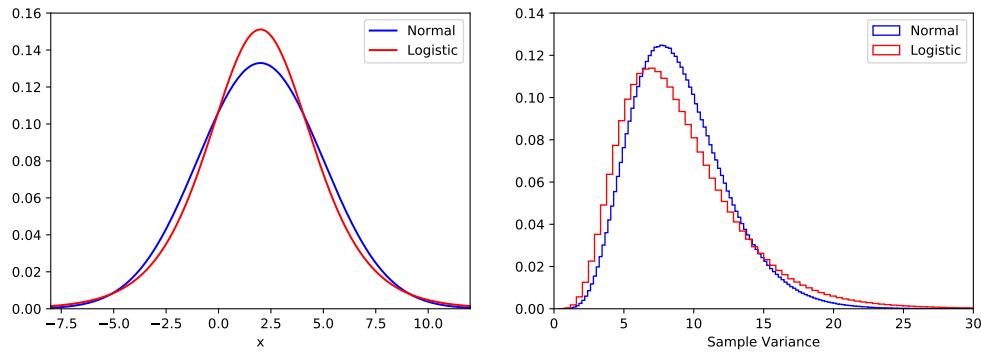


Figure 6.3: PDF's of the normal and Laplace distributions, along with histograms of the sample variances from the corresponding distributions.

```

2
3 std, n, N = 3, 15, 10^7
4 dNormal = Normal(2, std)
5 dLogistic = Logistic(2, sqrt(std^2*3)/pi)
6 xGrid = -8:0.1:12
7
8 sNormal = [var(rand(dNormal,n)) for _ in 1:N]
9 sLogistic = [var(rand(dLogistic,n)) for _ in 1:N]
10
11 figure(figsize=(12.4,4))
12 subplot(121)
13 plot(xGrid, pdf(dNormal,xGrid), "b",label="Normal")
14 plot(xGrid, pdf(dLogistic,xGrid), "r",label="Logistic")
15 xlabel("x")
16 legend(loc="upper right")
17 xlim(-8,12)
18 ylim(0,0.16)
19
20 subplot(122)
21 plt[:hist](sNormal,200,color="b",histtype="step",normed=true,
22 label="Normal")
23 plt[:hist](sLogistic,200,color="r",histtype="step",normed=true,
24 label="Logistic")
25 xlabel("Sample Variance")
26 legend(loc="upper right")
27 xlim(0,30)
28 ylim(0,0.14)

```

- In line 3 the number of sample observations, n , and total number of experiments, N , is specified.
- In lines 4-5 we define the two distributions with matched moments and variance. Note that the Julia `Logistic()` function uses the same parametrization as that of equation (6.18).
- In lines 8to 9 comprehensions are used to generate N sample variances from the normal and logistic distributions `dNormal` and `dLogistic`, with the values assigned to the variables `sLogistic` and `sNormal` respectively.
- In lines 13-14, the PDF's of the normal and Laplace distributions are plotted.

- In lines 20-24, a histogram of the sample variances of `sLogistic` is plotted, along with the analytically expected result, if the underlying process was normally distributed. It can be seen that the sample variances do not follow the expected result based on the normality assumption, and this suggest that the underlying process is not normally distributed. This example illustrates the fact that the distribution of sample variances is sensitive to the normality assumption.

Having seen that the distribution of the sample variance heavily depends on the shape of the actual distribution, we now investigate the effect that this has on the accuracy of the confidence interval. Specifically, usage of the confidence interval formula (6.17) does not yield coverage as in (6.16). We now investigate this further.

We cycle through different values of alpha from 0.001 to 0.1, and for each value, we perform N of the following identical experiments: calculate the sample variance of n observations and evaluate (6.17). We then calculate the proportion of times that the actual (unknown) variance of the distribution is contained within the confidence interval. These proportions (or α values) are then plotted against the actual values of α . See Figure 6.4. Note that only in the case of the normal distribution do the simulated α values align with those of the actual alphas used.

Listing 6.10: Actual α vs. α used in variance confidence intervals

```

1  using Distributions, PyPlot
2
3  std, n, N = 3, 15, 10^4
4  alphaUsed = 0.001:0.001:0.1
5  dNormal   = Normal(2, std)
6  dLogistic = Logistic(2, sqrt(std^2*3)/pi)
7
8  function alphaSimulator(dist, n, alpha)
9    popVar      = var(dist)
10   coverageCount = 0
11   for i in 1:N
12     sVar = var(rand(dist, n))
13     L = (n - 1) * sVar / quantile(Chisq(n-1), 1-alpha/2)
14     U = (n - 1) * sVar / quantile(Chisq(n-1), alpha/2)
15     coverageCount += L < popVar && popVar < U
16   end
17   1 - coverageCount/N
18 end
19
20 plot(alphaUsed, alphaSimulator.(dNormal,n,alphaUsed), ".b", label="Normal")
21 plot(alphaUsed, alphaSimulator.(dLogistic, n, alphaUsed), ".r", label="Logistic")
22 plot([0,0.1],[0,0.1], "k", lw=0.5)
23 xlabel("alpha used")
24 ylabel("alpha actual")
25 legend(loc="upper left")
26 xlim(0,0.1)
27 ylim(0,0.2)
```

- In lines 3-6 we define the number of observations in each group, and the total number of groups, n and N , along with the values of alpha we will use, `alphaUsed`, and the two distributions, `dNormal` and `dLogistic`.

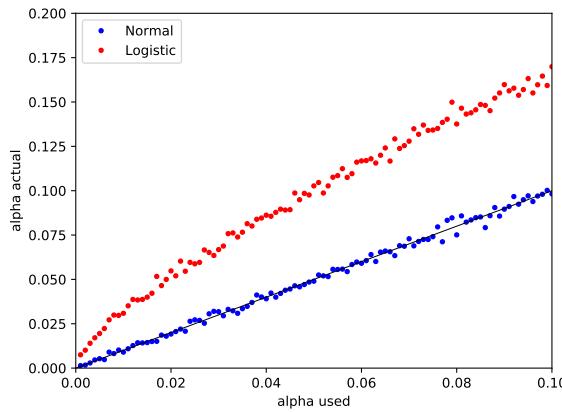


Figure 6.4: Comparison of actual α values vs. α values used in confidence intervals. Blue: normal population. Red: a logistic population.

- In lines 8-18, the function `alphaSimulator()` is defined. This function takes a distribution, the total number of sample observations, and a value of alpha as input, and generates N separate confidence intervals for the mean via equation (6.17). It then returns the corresponding proportion of times the confidence intervals do not contain the actual variance of the distribution.
- In lines 9-10, the variance of the distribution is stored as `popVar`, and the counter `coverageCount` initialized and set to zero.
- Lines 11-16 contain the main logic of this code block. In line 12 we calculate the sample variance of n observations from the distribution `dist`, and assign it to the variable `sVar`. In lines 13-14, equation (6.17) is implemented, and the lower and upper bounds of the confidence interval, `L` and `U`, calculated. Line 15 checks whether our population variance `popVar` is contained within the confidence interval $[L, U]$, and if it is, then `coverageCount` is incremented by 1. This process is repeated N times through the use of a `for` loop on line 11.
- In line 17, the proportion of confidence intervals that do not contain the population variance is returned. This number is analogous to a simulated value of α .
- In line 20 the function `alphaSimulated` is evaluated for all values of alpha in `alphaUsed` given a normal distribution, and the simulated alpha results are then plotted against the actual values of alpha.
- In line 20 the function `alphaSimulated` is evaluated for all values of alpha in `alphaUsed` given a logistic distribution, and the simulated alpha results are then plotted against the actual values of alpha.
- In line 22, a diagonal line is plotted, which represents the case where the simulated alpha values match those of the actual alpha values.
- The resulting Figure 6.4 shows that in the case of the normal distribution, the resulting simulated alpha values (i.e. coverage of the confidence intervals) are in general agreement with the actual alphas used. By contracts, in the case of the logistic distribution, the resulting simulated alpha values of confidence intervals do not correspond to the actual values of alpha

used. Instead, for non-small values of alpha in the logistic case, the simulated alpha values deviate out towards a constant value as alpha used increases. Note that for alphas around zero, the deviation is much smaller, reflecting the fact that, due to the width of the CI's and the number of simulations, almost all CI's contain the actual population variance.

6.5 Prediction Intervals

We now look at the concept of a *prediction interval*. A prediction interval tells us a predicted range that a single next observation of data is expected to fall within. This differs from a confidence interval which indicates us how confident we are of a particular parameter we are trying to estimate. The bounds of a prediction interval are always wider than those of a confidence interval, as the prediction interval must account for the uncertainty in knowing the population mean, as well as the scatter of the data due to variance.

Consider a sequence of data points x_1, x_2, x_3, \dots , which come from a normal distribution and are assumed i.i.d. Further assume that we observed x_1, \dots, x_n but have not yet observed X_{n+1} . In this case, a $100(1 - \alpha)\%$ prediction interval for the single future observation, X_{n+1} , is given by,

$$\bar{x} - t_{1-\frac{\alpha}{2}, n-1}s\sqrt{1 + \frac{1}{n}} \leq X_{n+1} \leq \bar{x} + t_{1-\frac{\alpha}{2}, n-1}s\sqrt{1 + \frac{1}{n}}, \quad (6.19)$$

where, \bar{x} and s are respectively the sample mean and sample standard deviation computed from x_1, \dots, x_n . Note that as the number of observations, n , increases, the bounds of the prediction interval decreases towards,

$$\bar{x} - z_{1-\frac{\alpha}{2}}s \leq X_{n+1} \leq \bar{x} + z_{1-\frac{\alpha}{2}}s. \quad (6.20)$$

In Listing 6.11 below we investigate prediction intervals based on a series of observations made from a normal distribution. In it, we start with $n = 2$ observations and calculate the corresponding prediction interval for the next observation. The sample size n is then progressively increased, and the prediction interval for each next observation calculated for each subsequent case.

Listing 6.11: Prediction interval given unknown population mean and variance

```

1  using Random, Statistics, Distributions, PyPlot
2  Random.seed!(3)
3
4  mu, sig = 50, 3
5  dist = Normal(mu, sig)
6  alpha = 0.01
7  N = 40
8
9  observations = rand(dist, 2)
10 ciLarray, ciUarray = [], []
11
12 for n in 2:N
13     xbar = mean(observations)
14     sd = std(observations)
15     tVal = quantile(TDist(n-1), 1-alpha/2)
16     delta = tVal * sd * sqrt(1+1/n)
17
18     ciL = xbar - delta

```

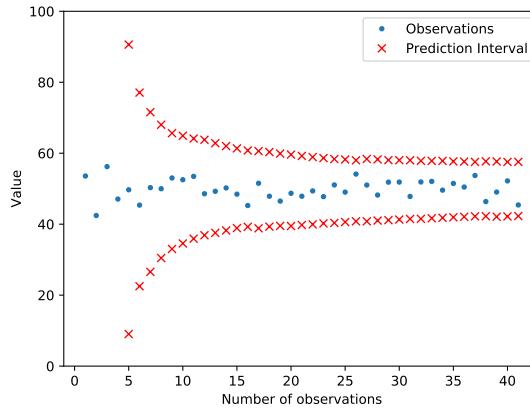


Figure 6.5: As the number of observations increases, the width of the prediction interval decreases to a constant.

```

19     ciU = xbar + delta
20
21     push! (ciLarray, ciL)
22     push! (ciUarray, ciU)
23
24     xNew = rand(dist)
25     push! (observations, xNew)
26 end
27
28 plot(1:N+1,observations,".",label="Observations")
29 plot(3:N+1,ciUarray,"rx",label="Prediction Interval")
30 plot(3:N+1,ciLarray,"rx")
31 ylim(0,100)
32 xlabel("Number of observations")
33 ylabel("Value")
34 legend(loc="upper right")

```

- In line 4-7 we setup our distribution, choose an alpha for our confidence interval, and also set the limiting number of observations we will make, N.
- In line 9 we sample our first two observations from our distribution, and store them in the array `observations`.
- In line 10, we create the arrays `ciLarray` and `CiUarray`, which will be used to store the lower and upper bounds of our confidence intervals respectively.
- Lines 12-26 contain the main logic of this example and in them the prediction intervals for each case of n observations, from n to N are calculated.
- In line 15 the T-statistic is calculated, given n observations (in the first instance this is 2).
- Then in lines 18-19, equation (6.19) is evaluated, and the lower and upper bounds of the prediction interval are stored in the arrays `ciLarray` and `CiUarray` respectively.
- Following this, in lines 24-25, we make a single new observation, which is then included in our set of sample observations `observations` for the next iteration. The loop then repeats.

- It can be observed from Figure 6.5 that as the number of observations increases, the prediction interval width decreases. Ultimately it follows (6.20). and has an expected width of $2 z_{1-\frac{\alpha}{2}} \sigma$.

6.6 Credibility Intervals

This section is under construction.

Chapter 7

Hypothesis Testing - DRAFT

This chapter explores hypothesis testing through a few specific practical hypothesis tests. To begin, recall the general hypothesis test formulation first introduced in Section 5.6. Denote,

$$H_0 : \theta \in \Theta_0, \quad H_1 : \theta \in \Theta_1.$$

Here Θ_0 and Θ_1 partition the parameter space Θ . One of the most common examples for a single population is to consider θ as μ , the population mean, in which case $\Theta = \mathbb{R}$. Often, we wish to test if the population mean is equal to some value, μ_0 , hence we can construct a *two sided hypothesis test* as follows,

$$H_0 : \mu = \mu_0, \quad H_1 : \mu \neq \mu_0. \quad (7.1)$$

However, one could instead chose to construct a *one sided hypothesis test*, as,

$$H_0 : \mu \leq \mu_0, \quad H_1 : \mu > \mu_0, \quad (7.2)$$

or alternatively, in the opposite direction,

$$H_0 : \mu \geq \mu_0, \quad H_1 : \mu < \mu_0, \quad (7.3)$$

where the choice of setting up (7.1), (7.2) or (7.3) depends on the context of the problem.

Once the hypothesis is established, the general approach involves calculating the *test statistic*, along with the corresponding *p-value*, and then finally making some statement about the null hypothesis based on some chosen level of significance. These concepts were first introduced in Section 5.6 and in this chapter we present some specific common examples often used in practice.

In Section 7.1 we introduce hypothesis testing via several examples involving a single population. In Section 7.2, we present extensions of these concepts and related ideas by looking at inference for the difference in means of two populations. In Section 7.3 we focus on methods of *Analysis of Variance (ANOVA)*, and then in Section 7.4 we explore *Chi-squared tests* and *Kolmogorov-Smirnov tests*. These latter two procedures are often used to assess goodness of fit and/or independence. We then close with Section 7.5, where we illustrate how power curves can aid in experimental design.

As in the previous chapters, we try to strike a balance between illustrating usage of the `HypothesisTests` package and reproducing the results from fundamental calculations, with the purpose of highlighting key phenomena. Several of the examples make use of the datasets `machine1.csv` and

`machine2.csv`, which represent the diameter (in mm) of bolts produced via two separate machines.

7.1 Single Sample Hypothesis Tests for the Mean

As an introduction, consider the case where we wish to make inference on the mean diameter of produced by a machine. Specifically, assume that we are interested in checking if the machine is producing bolts of the specified diameter $\mu_0 = 52.2$ mm. In this case, using a hypothesis testing methodology, we may wish to set-up the hypothesis as,

$$H_0 : \mu = 52.2, \quad H_1 : \mu \neq 52.2. \quad (7.4)$$

Here, H_0 represents the situation where the machine is functioning properly, and deviation from H_0 in either the positive or negative direction is captured by H_1 , which represents that the machine is malfunctioning. Alternatively, one could have treated $\mu_0 = 52.2$ as a specified upper limit on the bolt diameter, in which case the hypothesis would be formulated as,

$$H_0 : \mu \leq 52.2, \quad H_1 : \mu > 52.2. \quad (7.5)$$

Similarly, one could envision a case where (7.3) was used instead. In most of this chapter, we do not dive deeply into the aspects of formulating the hypotheses themselves (a key component of *experimental design*), but rather the hypotheses are introduced and treated as given.

Once the hypothesis is formulated, the next step is the collection of data, which in this section is taken from `machine1.csv`. We now separate the inference of the mean of a single population into the two cases of variance known and unknown, similarly to what was done in Section 6.1. Note that, similarly to Chapter 6, it is assumed that the observations, X_1, \dots, X_n are normally distributed. Finally, at the end of this section, we consider a simple *non-parametric* test, where we make no assumptions about the distribution of the observations.

Population Variance Known

Consider the case where we wish to test whether a single machine in the factory is producing bolts of a specified diameter. For this example, we set up the hypothesis as two sided according to (7.4), and assume that σ is a known value. Recall from Section 5.2 that, under H_0 , \bar{X} follows a normal distribution with mean μ_0 and variance σ^2/n . Hence it holds that under H_0 the *Z-statistic*,

$$Z = \frac{\bar{X} - \mu_0}{\sigma/\sqrt{n}}, \quad (7.6)$$

follows a standard normal distribution.

As we will see through the various examples in this chapter, the test statistic often follows a general form similar to that of (7.6). In this case, under the null hypothesis, the random variable Z is normally distributed, and hence to carry out a hypothesis test we observe its position relative to a standard normal distribution. Specifically, we check if it lies within the *rejection region* or not, and if it does, we reject the null hypothesis, otherwise we don't. In Figure 7.1 we present the rejection

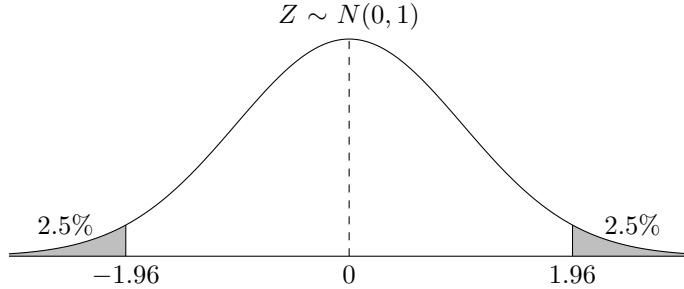


Figure 7.1: The standard normal distribution and rejection regions for the two sided hypothesis test (7.4) at significance level $\alpha = 5\%$.

region corresponding to $\alpha = 5\%$. It is obtained by considering the $\alpha/2$ and $1 - \alpha/2$ quantiles of the standard normal distribution.

With the hypothesis test and rejection region specified, we are ready to collect data, calculate the test statistic and make a conclusion. For this example, the data is taken from `machine1.csv` where we assume $\sigma = 1.2$ (known). After collecting the data, the observed z-statistic is calculated via,

$$z = \frac{\bar{x} - \mu_0}{\sigma/\sqrt{n}}. \quad (7.7)$$

We then reject H_0 if $|z| > z_{1-\alpha/2}$ where $z_{1-\alpha/2}$ is the quantile of the standard normal distribution for a specific confidence level α . We may also compute the p -value of the test via,

$$p = 2 \mathbb{P}(Z > |z|). \quad (7.8)$$

That is, we consider the observed test static, z , and determine the maximal significance level for which we would reject H_0 . Hence, for a fixed significance level α , if $p < \alpha$, we reject H_0 and otherwise not.

The calculation of critical values such as $z_{1-\alpha/2}$ and p -values is typically done via software, and more traditionally via *statistical tables*, which list the area under the normal curve along with different quantiles of a standard normal. For example $z_{0.025} = -1.96$, or $z_{0.975} = 1.96$. For $\alpha = 0.05$, we reject the null hypothesis if $z > 1.96$ or $z < -1.96$, otherwise we don't reject.

If the null hypothesis is rejected then we conclude the test by stating, “*there is sufficient evidence to reject the null hypothesis at the 5% significance level*”. Otherwise we conclude by stating, “*there is insufficient evidence to reject the null hypothesis at the 5% significance level*”.

If a different hypothesis test setup is used, such as (7.5), then the rejection region is not symmetric as in Figure 7.1, but rather covers only one tail of the distribution. This is illustrated in Figure 7.2, where $z_{0.95} = 1.64$ is used to determine the boundary of the rejection region. In such a case, the p -value is calculated via $p = \mathbb{P}(Z > z)$.

In Listing 7.1, we present an example containing two hypothesis tests (using the same data) where the first (`mu0A`) is rejected and the second (`mu0B`) is not-rejected. For the `mu0A` case, the test statistic is first calculated via (7.7) along with the corresponding p -value via (7.8). Then the `HypothesisTests` package is used to perform the same hypothesis test for both `mu0A` and `mu0B`.

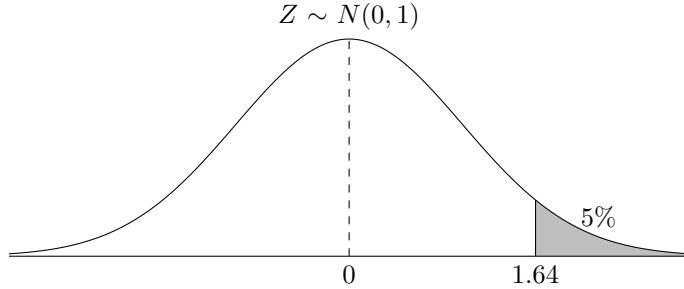


Figure 7.2: The standard normal distribution and rejection region for the one sided hypothesis test (7.5) at significance level $\alpha = 5\%$.

The default test assumes $\alpha = 5\%$. Observe that the p -value in the μ_0A case is less than 0.05 and hence H_0 is rejected. In comparison, for the μ_0B case, the p -value is greater than 0.05 and hence H_0 is not rejected.

Listing 7.1: Inference with single sample when population variance is known

```

1  using CSV, Distributions, HypothesisTests
2
3  data = CSV.read("machinel.csv", header=false)[:,1]
4  xBar, n = mean(data), length(data)
5  sigma = 1.2
6  mu0A, mu0B = 52.2, 53
7
8  testStatistic = (xBar - mu0A) / (sigma/sqrt(n))
9  pVal = 2*ccdf(Normal(), abs(testStatistic))
10
11 testA = OneSampleZTest(xBar, sigma, n, mu0A)
12 testB = OneSampleZTest(xBar, sigma, n, mu0B)
13
14 println("Results for mu0 = ", mu0A, ":")
15 println("Manually calculated test statistic: ", testStatistic)
16 println("Manually calculated p-value: ", pVal, "\n")
17 println(testA)
18
19 println("\n In case of mu0 = ", mu0B, " then p-value = ", pvalue(testB))

```

Results for $\mu_0 = 52.2$:

Manually calculated test statistic: 2.8182138203055467
 Manually calculated p-value: 0.004829163880878602

```

One sample z-test
-----
Population details:
    parameter of interest: Mean
    value under  $h_0$ :      52.2
    point estimate:        52.95620612127991
    95% confidence interval: (52.4303, 53.4821)

```

```

Test summary:
    outcome with 95% confidence: reject  $h_0$ 
    two-sided p-value:           0.0048

```

```
Details:
```

```
number of observations: 20
z-statistic: 2.8182138203055467
population standard error: 0.2683281572999747
```

```
In case of mu0 = 53 then p-value = 0.870352975060586
```

- In lines 3–6 we load the data, calculate the sample mean, and specify the values of μ_{0A} and μ_{0B} under H_0 (there are two separate tests in this code example). Note that importantly, the standard deviation, σ , is specified as 1.2, a value assumed ‘known’.
- In line 8 we calculate the test statistic for case μ_{0A} according to (7.7).
- In line 9 we calculate the p -value according to (7.8). Note that the `ccdf()` function is used to find the area to the right of the absolute value of the test statistic. The resulting p -value is then stored as the variable `pVal`.
- In lines 11 and 12, the `OneSampleZTest()` function from `HypothesisTests` is used to perform the same calculations. This is done for both the μ_{0A} and μ_{0B} case. The results are stored in `testA` and `testB`. These objects can then be printed or queried. Note that `OneSampleZTest()` was called with 4 arguments. If the last argument (μ_{0A} or μ_{0B}) was excluded, then the function would have performed the one sample z test assuming $\mu_0 = 0$. There is also an additional method for `OneSampleZTest()`, which simply takes a single argument of an array of values. In this case it will use the sample standard deviation as the population standard deviation, and will assume $\mu_0 = 0$. Further information is available in the documentation of the `HypothesisTests` package.
- Lines 14–17 print out results for the μ_{0A} case. As can be seen, the p -value of 0.0048 merits rejection of H_0 with $\alpha = 5\%$. The output from line 17 also lists the value of the parameter under H_0 , the point estimate of the parameter (`xBar`), as well as the corresponding 95% confidence interval.
- Line 19 prints the p -value for μ_{0B} , and since the p -value is greater than 0.05, we do not reject H_0 . Note the use of the `pvalue()` function applied to `testB`. This way of using the `HypothesisTests` package is based on creating an object (`testB` in this case) and then querying it using a function like `pvalue()`.

Population Variance Unknown

Having covered the case of variance known, we now consider the more realistic scenario where the population variance is unknown. Informally called the *T-test*, this is perhaps the most famous and widely used hypothesis test in elementary statistics. Here the test statistic is the *T-statistic*,

$$T = \frac{\bar{X} - \mu_0}{S/\sqrt{n}}. \quad (7.9)$$

Notice that it is similar to (7.6), however the sample standard deviation, S , is used instead of the population standard deviation σ , since σ is unknown. As presented in Section 5.2, in the case where

the data is normally distributed with mean μ_0 , the random variable T follows a T-distribution with $n - 1$ degrees of freedom and this is the basis for the *T-test*. The procedure is the same as the z-test presented above, except that a T-distribution is used instead of a normal. Note that for non-small n , the T-distribution is almost identical to a standard normal distribution.

The observed test statistic from the data is then

$$t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}}, \quad (7.10)$$

and the corresponding p -value for a two sided test is

$$p = 2 \mathbb{P}(T_{n-1} > |t|), \quad (7.11)$$

where T_{n-1} is a random variable distributed according to a T-distribution with $n - 1$ degrees of freedom. Note that standardized tables present *critical values* of the T-distribution, namely t_γ where γ is typically 0.9, 0.95, 0.975, 0.99 and 0.995. These are typically presented in detail for degrees of freedom ranging from $n = 2$ to $n = 30$, after which the T-distribution is very similar to a normal distribution. These values are then compared to the T-statistic (7.10) where $\gamma = 1 - \alpha$ in the one sided case or $\gamma = 1 - \alpha/2$ in the two sided case. However, for precise calculation of p -values, software must be used.

In Listing 7.2 below we first calculate the test statistic via (7.10), and then use this to manually calculate the corresponding p -value via (7.11). Then `OneSampleTTest()` from `HypothesisTests` is used to perform the same calculations.

Listing 7.2: Inference with single sample when population variance unknown

```

1  using CSV, Distributions, HypothesisTests
2
3  data = CSV.read("machine1.csv", header=false, allowmissing=:none)[:,1]
4  xBar, n = mean(data), length(data)
5  s = std(data)
6  mu0 = 52.2
7
8  testStatistic = (xBar - mu0) / (s/sqrt(n))
9  pVal = 2*ccdf(TDist(n-1), abs(testStatistic))
10
11 println("Manually calculated test statistic: ", testStatistic)
12 println("Manually calculated p-value: ", pVal)
13 OneSampleTTest(data, mu0)

```

```

One sample t-test
-----
Population details:
    parameter of interest: Mean
    value under h_0:      52.2
    point estimate:       52.95620612127991
    95% confidence interval: (52.4039, 53.5085)

Test summary:
    outcome with 95% confidence: reject h_0
    two-sided p-value:          0.0099

Details:

```

```

number of observations: 20
t-statistic: 2.86553950269453
degrees of freedom: 19
empirical standard error: 0.2638965962845154

Manually calculated test statistic: 2.86553950269453
Manually calculated p-value: 0.009899631865162935

```

- In lines 3-6 the data is loaded, then the sample mean calculated, and the value of μ_0 under H_0 specified. Note that the sample standard deviation is calculated and stored as s .
- In line 8 the test statistic is calculated according to (7.10).
- In line 9 the p -value is calculated according to (7.11). Note that the `ccdf()` function is used on a T-distribution with $n - 1$ degrees of freedom, `TDist(n-1)`. The p -value is then stored as the variable `pVal`.
- In lines 11 and 12 the results of our manually calculated test statistic and corresponding p -value are printed.
- In line 13 the function `OneSampleTTest()` is used to perform the test on the data. Note that in this case we only specify two arguments, the array of our data, and the value of μ_0 , `mu0`. We can see that the output from our manual calculation matches that of the `OneSampleTTest()` function.

A Non-parametric Sign Test

The validity of the T-test relies heavily on the assumption that the sample X_1, \dots, X_n is comprised of independent normal random variables. That is, only under this assumption, does the T-statistic follow a T-distribution. This assumption may often be safely made, however in certain cases we cannot assume a normal population.

For such cases one needs an alternative test. For this we present a particular type of *non-parametric test*, known as the *sign test*. Here the phrase “non-parametric” implies that the distribution of the test statistic does not depend on any particular distributional assumption for the population.

For the sign test, we begin by denoting the random variables,

$$X^+ = \sum_{i=1}^n \mathbf{1}\{X_i > \mu_0\} \quad \text{and} \quad X^- = \sum_{i=1}^n \mathbf{1}\{X_i < \mu_0\} = n - X^+. \quad (7.12)$$

where $\mathbf{1}\{\cdot\}$ is the indicator function. The variable X^+ is a count of the number of observations that exceed μ_0 , and similarly, X^- is a count of the number of observations that are below μ_0 .

Observe now that under $H_0 : \mu = \mu_0$ it holds that $\mathbb{P}(X_i > \mu_0) = \mathbb{P}(X_i < \mu_0) = 1/2$. Note that we are actually taking here μ_0 as the median of the distribution and assuming that $\mathbb{P}(X_i = \mu_0) = 0$.

Now, under H_0 , the random variables, X^+ and X^- both follow a binomial($n, 1/2$) distribution (see Section 3.5). Given the symmetry of this binomial distribution we define the test statistic to be,

$$U = \max\{X^+, X^-\}. \quad (7.13)$$

Hence with observed data, and an observed test statistic u , the p -value can be calculated via,

$$p = 2 \mathbb{P}(B > u), \quad (7.14)$$

where B is a binomial($n, 1/2$) random variable. Here, under H_0 , p is the probability of getting an extreme number of signs greater than u (either too many via X^+ or a very small number via X^-). The test procedure is then to reject H_0 if $p < \alpha$.

In Listing 7.3 below we present an example where we calculate the value of the test statistic and its corresponding p -value manually. We then use these to make conclusions about the null hypothesis at the 5% significance level. As was done in Listing 7.1 we compare two hypothetical cases. In the first case $\mu_0 = 52.2$, and the second $\mu_0 = 53.0$. As can be observed from the output, the former case is significant (H_0 is rejected) while the latter is not.

Listing 7.3: Non-parametric sign test

```

1  using CSV, Distributions, HypothesisTests
2
3  data = CSV.read("machine1.csv", header=false, allowmissing=:none)[:,1]
4  n = length(data)
5  mu0A, mu0B = 52.2, 53
6
7  xPositiveA = sum(data .> mu0A)
8  testStatisticA = max(xPositiveA, n-xPositiveA)
9
10 xPositiveB = sum(data .> mu0B)
11 testStatisticB = max(xPositiveB, n-xPositiveB)
12
13 binom = Binomial(n, 0.5)
14 pValA = 2*ccdf(binom, testStatisticA)
15 pValB = 2*ccdf(binom, testStatisticB)
16
17 println("Binomial mean: ", mean(binom))
18
19 println("Case A: mu0: ", mu0A)
20 println("\tTest statistic: ", testStatisticA)
21 println("\tP-value: ", pValA)
22
23 println("Case B: mu0: ", mu0B)
24 println("\tTest statistic: ", testStatisticB)
25 println("\tP-value: ", pValB)

```

```

Binomial mean: 10.0

Case A: mu0: 52.2
        Test statistic: 15
        P-value: 0.011817932128906257
Case B: mu0: 53
        Test statistic: 11
        P-value: 0.5034446716308596

```

- In lines 3 to 5 the data is loaded and the value of the population mean under the null hypothesis for both cases, μ_0A and μ_0B , specified.
- In lines 7 to 11 the observed test statistics for both cases are calculated via (7.13). Note the use of `.>` for comparing the array data element wise with the scalars μ_0A and μ_0B .
- In lines 13 to 15, the `Binomial()` function from the `Distributions` package is used to create a binomial distribution of size n and probability 0.5. This is then used to compute the p -values for both cases in lines 14 and 15 via (7.14). Note the use of the `ccdf()` function.
- The results are printed in lines 19 to 25. As there are $n = 20$ observations the binomial mean is 10. A test statistic of 15 (as is the case for $\mu_0 = 52.2$) yields a small p -value and we reject H_0 for $\alpha = 0.05$. However, for $\mu_0 = 53$ the test statistic of $u = 11$ is not non-plausible under H_0 and hence we don't reject H_0 .

Sign Test vs. T-Test

With the sign test presented as a robust alternative to the T-test, one may ask why not simply always use the sign test. After all, the validity of the T-test rests on the assumption that X_1, \dots, X_n are normally distributed. Otherwise, T of (7.9) does not follow a T-distribution, and conclusions drawn from the test may be potentially imprecise.

One answer is statistical power. As we show in the example below, the T-test is a more powerful test than the sign test when the normality assumption holds. That is, if the normality assumption can be safely made, then the power of the T-test exceeds that of the sign-test. That is, for a fixed α , the probability of detecting H_1 is higher for the T-test than for the sign test. This makes it a more effective test to use, if the data can be assumed normally distributed. The concept of power was first introduced in Section 5.6, and is further investigated in Section 7.5

In Listing 7.4 we perform a two-sided hypothesis test for $H_0 : \mu = 53$ vs. $H_1 : \mu \neq 53$ via both the T-test and sign test. We consider a range of scenarios where we let the actual μ vary over $[51.0, 55.0]$. When $\mu = 53$, H_0 is the case, however all other cases fall in H_1 . On a grid of such cases we use Monte Carlo to estimate the power of the test (for $\sigma = 1.2$). The resulting curves in Figure 7.3 show that the T-test is more powerful than the sign test.

Listing 7.4: Comparison of sign test and T-test

```

1  using Random, Distributions, HypothesisTests, PyPlot
2
3  muRange = 51:0.02:55
4  n = 20
5  N = 10^4
6  mu0 = 53.0
7
8  powerT, powerU = [], []
9
10 for muActual in muRange
11
12     dist = Normal(muActual, 1.2)
13     rejectT, rejectU = 0, 0
14     Random.seed!(1)

```

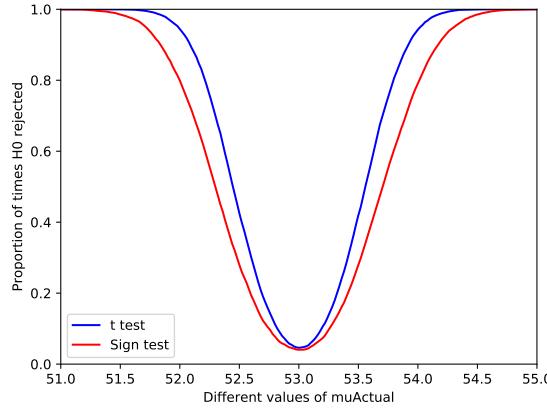


Figure 7.3: Power of the T-test vs. power of the sign-test.

```

15
16     for _ in 1:N
17         data = rand(dist,n)
18         xBar, stdDev = mean(data), std(data)
19
20         tStatT = (xBar - mu0) / (stdDev/sqrt(n))
21         pValT   = 2*ccdf(TDist(n-1), abs(tStatT))
22
23         xPositive = sum(data .> mu0)
24         uStat = min(xPositive, n-xPositive)
25         pValSign = 2*cdf(Binomial(n,0.5), uStat)
26
27         rejectT += pValT < 0.05
28         rejectU += pValSign < 0.05
29     end
30
31     push! (powerT, rejectT/N)
32     push! (powerU, rejectU/N)
33
34 end
35
36 plot(muRange, powerT, "b",label="t test");
37 plot(muRange, powerU, "r",label="Sign test");
38 xlim(51,55)
39 ylim(0,1)
40 legend(loc="bottom right")
41 xlabel("Different values of muActual")
42 ylabel("Proportion of times H0 rejected")

```

- In lines 3–6 we setup the basic parameters. The sample size n is 20. The range μ represents the range $[51.0, 55.0]$ in discrete steps of 0.02. The number N is the number of simulation repetitions to carry out for each value of μ in that range. The value μ_0 is the value under H_0 .
- In line 8 we initialize empty arrays that are to be populated with power estimates.
- Lines 10–34 contain the main loop where for each iteration another value, μ_{Actual} is tested. In line 12, for each such value there is another distribution dist , each time with the same

standard deviation, 1.2, but with a different mean, `muActual`. The inner loop of lines 16–29 is a repetition of the sampling experiment `N` times where for the same data we compute both `tStat` and `uStat` and the corresponding *p*-values. Rejection counts are accumulated in lines 27 and 28. Every time `pValT < 0.05` or similarly `pValSign < 0.05` constitutes a rejection.

- Note the use of `Random.seed!()` in Line 14 which allows us to obtain a smoother curve via common random numbers. See also Section 10.6.
- Lines 31–32 record the power for the respective `muActual` by appending to the arrays using `push!()`.
- Lines 36–42 plot the power curves showing the superiority of the T-test. Observe that at $\mu = 53$ the power is identical to $\alpha = 0.05$.

7.2 Two Sample Hypothesis Tests for Comparing Means

Having dealt with several examples involving one population, in this section we present some common hypothesis tests for the inference on the difference in means of two populations. As with all hypothesis tests, we start by first establishing the testing methodology. Commonly we wish to investigate if the population difference, Δ_0 , takes on a specific value. Hence we may wish to set up a two sided hypothesis test as,

$$H_0 : \mu_1 - \mu_2 = \Delta_0, \quad H_1 : \mu_1 - \mu_2 \neq \Delta_0. \quad (7.15)$$

Alternatively, one could formulate a one sided hypothesis test, such as,

$$H_0 : \mu_1 - \mu_2 \leq \Delta_0, \quad H_1 : \mu_1 - \mu_2 > \Delta_0, \quad (7.16)$$

or the reverse if desired. It is common to consider $\Delta_0 = 0$, in which case (7.15) would be stated as $H_0 : \mu_1 = \mu_2$, and similarly (7.16) as $H_0 : \mu_1 \leq \mu_2$. Once the testing methodology has been established, the approach then follows the same outline as that covered previously, the test statistic is calculated, along with its corresponding *p*-value, and then used to make some conclusion about the hypothesis for some significance level α .

For the tests introduced in this section we assume that the observations $X_1^{(1)}, \dots, X_{n_1}^{(1)}$ from population 1 and $X_1^{(2)}, \dots, X_{n_2}^{(2)}$ from population 2 are all normally distributed, where $X_i^{(j)}$ has mean μ_j and variance σ_j^2 . The testing methodology then differs based on the following three cases,

- (I) The population variances σ_1 and σ_2 are known.
- (II) The population variances σ_1 and σ_2 are unknown, but considered equal.
- (III) The population variances σ_1 and σ_2 are unknown, and considered unequal.

In each of these cases, the test statistic is given by,

$$\frac{\bar{X}_1 - \bar{X}_2 - \Delta_0}{S_{\text{err}}}, \quad (7.17)$$

where \bar{X}_j is the sample mean of $X_1^{(j)}, \dots, X_{n_j}^{(j)}$, and the standard error S_{err} varies according to the case (**I–III**). In each example, the two datasets `machine1.csv` and `machine2.csv` are used and it is considered that H_0 implies that both machines are identical (i.e. we use (7.15) with $\Delta_0 = 0$).

Population Variances Known

In case (**I**), where the population variances σ_1^2 and σ_2^2 are known, we set,

$$S_{\text{err}} = \sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}.$$

In this case, S_{err} is not a random quantity, and hence the test statistic (7.17) follows a standard normal distribution under H_0 . This is due to the distribution of \bar{X}_j as described in Section 5.2. Hence with the data at hand, the observed test statistic is,

$$z = \frac{(\bar{x}_1 - \bar{x}_2) - \Delta_0}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}.$$
 (7.18)

At this point, z is used for hypothesis tests in a manner analogous to the single sample test for the population mean when the variance known, as described at the start of Section 7.1.

Note that in reality, it is highly unlikely that both the population variances would be known, hence the `HypothesisTests` package does not contain functionality for this test. Nevertheless, in Listing 7.5 below, we perform the hypothesis test manually for pedagogical completeness.

Listing 7.5: Inference on difference of two means (variances known)

```

1  using CSV, Distributions, HypothesisTests
2
3  data1 = CSV.read("machine1.csv", header=false, allowmissing=:none)[:,1]
4  data2 = CSV.read("machine2.csv", header=false, allowmissing=:none)[:,1]
5  xBar1, n1 = mean(data1), length(data1)
6  xBar2, n2 = mean(data2), length(data2)
7  sig1, sig2 = 1.2, 1.6
8  delta0 = 0
9
10 testStatistic = (xBar1-xBar2 - delta0) / (sqrt(sig1^2 / n1 + sig2^2 / n2))
11 pVal = 2*ccdf(Normal(), abs(testStatistic))
12
13 println("Sample mean machine 1: ", xBar1)
14 println("Sample mean machine 2: ", xBar2)
15 println("Manually calculated test statistic: ", testStatistic)
16 println("Manually calculated p-value: ", pVal)

```

```

Sample mean machine 1: 52.95620612127991
Sample mean machine 2: 50.94739671468099
Manually calculated test statistic: 4.340167327618076
Manually calculated p-value: 1.423742605667141e-5

```

- In lines 3 to 7 we load our data, calculate the sample means, and specify the values of the population variances.

- In line 8, we specify the value of our test parameter under the null hypothesis, `delta0`, as 0.
- In line 10 we calculate the test statistic via (7.18).
- In line 10 we calculate the *p*-value.
- As the output shows, there is a very significant difference between the machines with the *p*-value almost zero.

Variance Unknown, but Assumed Equal

We now consider case **(II)** where the population variances are unknown, and assumed equal ($\sigma^2 := \sigma_1^2 = \sigma_2^2$). In this case the pooled sample variance, S_p^2 , is used to estimate σ^2 based on both samples. As covered in Section 6.2, it is given by,

$$S_p^2 = \frac{(n_1 - 1)S_1^2 + (n_2 - 1)S_2^2}{n_1 + n_2 - 2}, \quad (7.19)$$

where S_j^2 is the sample variance of sample j . It can be shown that under H_0 , if we set,

$$S_{\text{err}} = S_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}},$$

the test statistic is distributed according to a T-distribution with $n_1 + n_2 - 2$ degrees of freedom. Hence with the data at hand, the observed test statistic is,

$$t = \frac{(\bar{x}_1 - \bar{x}_2) - \Delta_0}{s_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}}, \quad (7.20)$$

where s_p is the observed pooled sample variance. At this point the procedure follows similar lines to the single sample T-test described above. Note that this two sample T-test with equal variance, is one of the most commonly used tests in statistics.

In Listing 7.6 we present an example where we perform a two sided hypothesis test on the difference in means of bolts produced from machines 1 and 2. First the test is performed manually, and then the `EqualVarianceTTest()` function from the `HypothesisTests` package is used.

Listing 7.6: Inference on difference of means (variances unknown, assumed equal)

```

1  using CSV, Distributions, HypothesisTests
2
3  data1 = CSV.read("machine1.csv", header=false, allowmissing=:none)[:,1]
4  data2 = CSV.read("machine2.csv", header=false, allowmissing=:none)[:,1]
5  xBar1, s1, n1 = mean(data1), std(data1), length(data1)
6  xBar2, s2, n2 = mean(data2), std(data2), length(data2)
7  delta0 = 0
8
9  sP = sqrt( ( (n1-1)*s1^2 + (n2-1)*s2^2 ) / (n1 + n2 - 2) )
10 testStatistic = (xBar1-xBar2 - delta0) / (sP * sqrt(1/n1 + 1/n2))
11 pVal = 2*ccdf(TDist(n1+n2 -2), abs(testStatistic))
12
13 println("Manually calculated test statistic: ", testStatistic)
14 println("Manually calculated p-value: ", pVal)
15 println(EqualVarianceTTest(data1, data2, delta0))

```

```

Manually calculated test statistic: 4.5466542394674425
Manually calculated p-value: 5.9493058655043084e-5

Two sample t-test (equal variance)
-----
Population details:
  parameter of interest: Mean difference
  value under h_0: 0
  point estimate: 2.008809406598921
  95% confidence interval: (1.1128, 2.9049)

Test summary:
  outcome with 95% confidence: reject h_0
  two-sided p-value: <1e-4

Details:
  number of observations: [20,18]
  t-statistic: 4.5466542394674425
  degrees of freedom: 36
  empirical standard error: 0.44182145832893077

```

- In lines 3–6 we load our data and then calculate the sample means, the sample variances and set the sample sizes, n_1 and n_2 .
- In line 7 we specify the value of our test parameter under the null hypothesis, δ_{00} , as 0.
- In line 9 we calculate the square root of the pooled sample variance, s_P , from (7.19). Note the use of the `sqrt()` function, as the test statistic, (7.20), makes use of s_p not s_p^2 .
- In lines 10 to 14 the test statistic is calculated via (7.20), and in line 11 the corresponding p -value, $pVal$, is calculated. These two values are then printed in lines 13 and 14.
- In line 15 the `EqualVarianceTTest()` function is used to perform the hypothesis test. Note that we give the function three arguments, first the two arrays, `data1` and `data2`, and the final argument the value of the test parameter under H_0 . Note that we did not need to specify the last argument in this case, as the function defaults to $\Delta_0 = 0$ if only two arguments are given. However here we specify three arguments to show the reader the general use of the function.
- It can be seen from the resulting output that the test statistic and p -value calculated manually match those calculated via the `EqualVarianceTTest()` function.

Variance Unknown, but not Assumed Equal

In case (III) where the population variances are unknown, and not assumed equal ($\sigma_1^2 \neq \sigma_2^2$), we set

$$S_{\text{err}} = \sqrt{\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}}$$

Then the observed test statistic is given by

$$t = \frac{(\bar{x}_1 - \bar{x}_2) - \Delta_0}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}. \quad (7.21)$$

As covered in Section 6.2, the distribution of the test statistic does not follow an exact T-distribution. Instead, we use the *Satterthwaite approximation*, and determine the degrees of freedom via,

$$v = \frac{(s_1^2/n_1 + s_2^2/n_2)^2}{\frac{(s_1^2/n_1)^2}{n_1-1} + \frac{(s_2^2/n_2)^2}{n_2-1}}. \quad (7.22)$$

In Listing 7.7 below, we perform a two sided hypothesis test that the difference between the population means is zero ($\Delta_0 = 0$). We first manually calculate the test statistic and corresponding *p*-value, and then make use of the `UnequalVarianceTTest()` function from the `HypothesisTests` package.

Listing 7.7: Inference on difference of means (variances unknown, assumed unequal)

```

1  using CSV, Distributions, HypothesisTests
2
3  data1 = CSV.read("machine1.csv", header=false, allowmissing=:none)[:,1]
4  data2 = CSV.read("machine2.csv", header=false, allowmissing=:none)[:,1]
5  xBar1, xBar2 = mean(data1), mean(data2)
6  s1, n1 = std(data1), length(data1)
7  s2, n2 = std(data2), length(data2)
8  delta0 = 0
9
10 v = (s1^2/n1 + s2^2/n2)^2 / ((s1^2/n1)^2/(n1-1) + (s2^2/n2)^2/(n2-1))
11 testStatistic = (xBar1-xBar2 - delta0) / sqrt(s1^2/n1 + s2^2/n2)
12 pVal = 2*ccdf(TDist(v), abs(testStatistic))
13
14 println("Manually calculated degrees of freedom, v: ", v)
15 println("Manually calcualted test statistic: ", testStatistic)
16 println("Manually calculated p-value: ", pVal)
17 println(UnequalVarianceTTest(data1, data2, delta0))

```

```

Manually calculated degrees of freedom, v: 31.82453144280283
Manually calcualted test statistic: 4.483705005611673
Manually calculated p-value: 8.936189820683007e-5
Two sample t-test (unequal variance)
-----
```

```

Population details:
    parameter of interest: Mean difference
    value under h_0:          0
    point estimate:           2.008809406598921
    95% confidence interval: (1.096, 2.9216)
```

```

Test summary:
    outcome with 95% confidence: reject h_0
    two-sided p-value:           <1e-4
```

```
Details:
```

```

number of observations: [20,18]
t-statistic: 4.483705005611673
degrees of freedom: 31.82453144280282
empirical standard error: 0.4480244360600785

```

- In lines 3 to 8 we load our data, calculate the sample means, sample standard deviations, and set the value of the test parameter, `delta0`, under H_0 to zero.
- In line 10 we calculate the degrees of freedom, `v`, via equation (7.22).
- In line 11 we calculate the test statistic via equation (7.21), and in line 12 we calculate the corresponding *p*-value, `pVal`.
- In line 17 we use the `UnequalVarianceTTest()` function to perform the hypothesis test, and it can be seen that the output matches the manually calculated values of the degrees of freedom, test statistic and *p*-value.

7.3 Analysis of Variance (ANOVA)

The methods presented in Section 7.2 handle the problem of comparing means of two populations. However, what if there were more than two populations that needed to be compared? This is often the case in biological, agricultural and medical trials, among other fields, where it is of interest to see if various “treatments” have an effect on some mean value or not. In these cases each type of treatment is considered as a different population.

More formally, assume that there is some *overall mean* μ and there are $L \geq 2$ treatments, where each treatment may potentially alter the mean by τ_i . In this case, the mean of the population under treatment i can be represented by $\mu_i = \mu + \tau_i$, where,

$$\sum_{i=1}^L \tau_i = 0.$$

This condition on the parameters $\{\tau_i\}$ ensures that given μ_1, \dots, μ_L , the overall μ and $\{\tau_i\}$ are well defined.

The question is then: “do the treatments have any effect or not”. This is then presented via the hypothesis formulation:

$$H_0 : \tau_1 = \tau_2 = \dots = \tau_L = 0, \quad \text{vs.} \quad H_1 : \exists i \mid \tau_i \neq 0. \quad (7.23)$$

Notice that H_0 is equivalent to the statement that $\mu_1 = \dots = \mu_L$, indicating that the treatments do not have an effect. Furthermore, H_1 is equivalent to the case where there exist at least two treatments, i and j such that $\mu_i \neq \mu_j$. In other words this means that the choice of treatment has an effect, at least between some treatments.

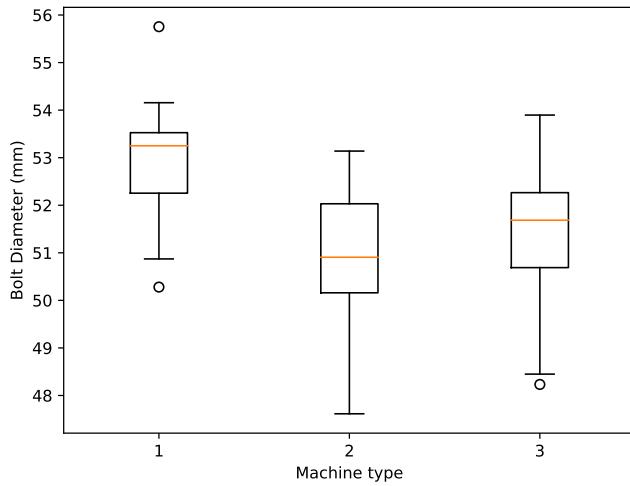


Figure 7.4: Box-plots of diameters associated with machines 1, 2 and 3.

In conducting hypotheses such as (7.23), we collect observations (data) as follows,

$$\begin{aligned} \text{Treatment 1: } & x_{11}, x_{12}, \dots, x_{1n_1}, \\ \text{Treatment 2: } & x_{21}, x_{22}, \dots, x_{2n_2}, \\ & \vdots \\ \text{Treatment L: } & x_{L1}, x_{12}, \dots, x_{Ln_L}, \end{aligned}$$

where n_1, \dots, n_L are the sample sizes for the treatments. If all samples are the same size (say $n_j = n$) then this is called a *balanced design* problem. However, often different treatments have different sample sizes, hence it is convenient to denote the total number of observations via,

$$m = \sum_{j=1}^L n_j. \quad (7.24)$$

Note that in a balanced design we have $m = L n$.

Once the data is collected, a common first step is to plot the observations. This is often done via *Box-plots* (see Section 4.3) as in Figure 7.4. This figure is generated by Listing 7.8, where the two diameter files used in the previous example are considered, along with a third data file, `machine3.csv`. Note that in this example the different machines are considered as different treatments, or *levels*.

Listing 7.8: Box-plots of data

```

1  using CSV, PyPlot
2
3  data1 = CSV.read("machine1.csv", header=false, allowmissing=:none)[:,1]
4  data2 = CSV.read("machine2.csv", header=false, allowmissing=:none)[:,1]
5  data3 = CSV.read("machine3.csv", header=false, allowmissing=:none)[:,1]
6
7  boxplot([data1,data2,data3])
8  xlabel("Machine type")

```

```

9   ylabel("Bolt Diameter (mm)")
10  println("n1 = ",length(data1),",\tn2 = ",length(data2), ",\tn3 = ",length(data3))

```

```
n1 = 20,           n2 = 18,           n3 = 18
```

While Figure 7.4 gives some indication of potential differences between the groups, it is not straightforward to quantify this via visual inspection. Hence the next step is to consider the values of the sample means for each individual treatment,

$$\bar{x}_j = \frac{1}{n_j} \sum_{i=1}^{n_j} x_{ij}.$$

These values can then be compared with the overall sample mean,

$$\bar{x} = \frac{1}{m} \sum_{j=1}^L \sum_{i=1}^{n_j} x_{ij} = \sum_{j=1}^L \frac{n_j}{m} \bar{x}_j. \quad (7.25)$$

In Listing 7.9 the sample means are computed. Note the use of the broadcast operator on line 3.

Listing 7.9: Sample means for ANOVA

```

1  using CSV
2
3  rfile(name) = CSV.read(name, header=false, allowmissing=:none)[:,1]
4  data = rfile.(["machine1.csv","machine2.csv","machine3.csv"])
5  println("Sample means for each treatment: ",round.(mean.(data),digits=2))
6  println("Overall sample mean: ",round(mean(vcat(data...)),digits=2))

```

```
Sample means for each treatment: [52.96, 50.95, 51.43]
Overall sample mean: 51.82
```

Although the mean values (52.96, 50.95 and 51.43) are needed, observing them on their own does not conclusively establish whether or not the treatments (machines) affect the response (diameter). The typical way to establish whether or not an effect between the groups does exist is to examine the variability of the individual treatment means, and compare these to the overall variability of the observations. If the variability of means significantly exceeds the variability of the individual observations, then H_0 is rejected, otherwise it is not.

This approach is called *ANOVA*, which stands for *analysis of variance*, and it is based on the decomposition of the sum of squares. In fact ANOVA is a broad collection of statistical methods, and here we only provide an introduction to ANOVA by covering the *one-way anova* test. In this test, the statistical model assumes that the observations of each treatment group come from an underlying model of the following form,

$$X_j = \mu_j = \mu + \tau_j + \varepsilon \quad \text{where} \quad \varepsilon \sim N(0, \sigma^2), \quad (7.26)$$

where X_j is the model for the j th treatment group and ε is some noise term with common unknown variance across all treatment groups, independent across measurements. In this sense, the ANOVA model (7.26) generalizes the assumptions of the T-test applied to case II (comparison of

two population means with variance unknown and assumed equal), as presented in the previous section.

The process of conducting a *one-way ANOVA test* follows the same general approach as any other hypothesis test. First the test statistic is calculated, then the corresponding *p-value*, and finally the *p-value* is used to make some conclusion about whether or not to reject H_0 at some chosen confidence level α . The test statistic for ANOVA is known as the *F-value*, and is the ratio of the average variance between the groups, divided by the average variance within the groups. Under the null hypothesis, F is distributed according to the *F distribution*, covered at the end of Section 5.2. Hence the ANOVA test is sometimes referred to as the *F-test*.

We now present the mathematical motivation used to calculate the variability within the groups and the variability between the groups through a simple example.

Decomposing Sum of Squares

A key idea of ANOVA is the decomposition of the total variability into two components: the variability between the treatments, and the variability within the treatments. There are explicit expressions for both, and here we show how to derive them by performing what is known as the *decomposition of the sum of squares*.

The total variance, also known as the *sum of squares total* (SS_{Total}), is a measure of the total variability of all observations, and is calculated as follows,

$$SS_{\text{Total}} = \sum_{j=1}^L \sum_{i=1}^{n_j} (x_{ij} - \bar{x})^2, \quad (7.27)$$

where \bar{x} is given by (7.25). Now through algebraic manipulation (adding and subtracting treatment means) we can show that SS_{Total} can be decomposed as follows,

$$\begin{aligned} \sum_{j=1}^L \sum_{i=1}^{n_j} (x_{ij} - \bar{x})^2 &= \sum_{j=1}^L \sum_{i=1}^{n_j} (x_{ij} - \bar{x}_j + \bar{x}_j - \bar{x})^2 \\ &= \sum_{j=1}^L \sum_{i=1}^{n_j} \left((x_{ij} - \bar{x}_j)^2 - 2(x_{ij} - \bar{x}_j)(\bar{x}_j - \bar{x}) + (\bar{x}_j - \bar{x})^2 \right) \\ &= \sum_{j=1}^L \sum_{i=1}^{n_j} (x_{ij} - \bar{x}_j)^2 + \sum_{j=1}^L n_j (\bar{x}_j - \bar{x})^2. \end{aligned} \quad (7.28)$$

Note that on the second line, the middle term reduces to zero, since $\sum_{i=1}^{n_j} (x_{ij} - \bar{x}_j) = 0$. Hence we have shown that the total variance, SS_{Total} , can be decomposed to,

$$SS_{\text{Total}} = SS_{\text{Error}} + SS_{\text{Treatment}}, \quad (7.29)$$

where,

$$SS_{\text{Error}} = \sum_{j=1}^L \sum_{i=1}^{n_j} (x_{ij} - \bar{x}_j)^2 \quad \text{and} \quad SS_{\text{Treatment}} = \sum_{j=1}^L n_j (\bar{x}_j - \bar{x})^2. \quad (7.30)$$

Note that the *sum of squares error*, SS_{Error} , is also known as the sum of the variability within the groups, and that the *sum of squares Treatment*, $SS_{\text{Treatment}}$, is also known as the variability between the groups. The decomposition (7.29) holds under both H_0 and H_1 , and hence allows us to construct a test statistic. Under H_0 , both SS_{Error} and $SS_{\text{Treatment}}$ should contribute to SS_{Total} in the same manner (once properly normalized). Alternatively, under H_1 it is expected that $SS_{\text{Treatment}}$ would contribute more heavily to the total variability.

Before proceeding with the construction of a test statistic, we present Listing 7.10, where the decomposition of (7.29) is demonstrated, for the purpose of showing how to compute its individual components in Julia. Note that this verification of the decomposition is not something one would normally carry out in practice as it is already proven in (7.27).

Listing 7.10: Decomposing the sum of squares

```

1  using Random
2  Random.seed!(1)
3  x1Dat = rand(24)
4  x2Dat = rand(15)
5  x3Dat = rand(73)
6
7  allData = [x1Dat,x2Dat,x3Dat]
8  xBarArray = mean.(allData)
9  nArray = length.(allData)
10 xBarTotal = mean(vcat(allData...))
11 L = length(nArray)
12
13 ssBetween =
14     sum([nArray[i]*(xBarArray[i] - xBarTotal)^2 for i in 1:L])
15 ssWithin =
16     sum([sum([(ob - xBarArray[i])^2 for ob in allData[i]]) for i in 1:L])
17 ssTotal =
18     sum([sum([(ob - xBarTotal)^2 for ob in allData[i]]) for i in 1:L])
19
20 println("Sum of squares between groups: ", ssBetween)
21 println("Sum of squares within groups: ", ssWithin)
22 println("Sum of squares total: ", ssTotal)

```

```

Sum of squares between groups: 0.2941847110381936
Sum of squares within groups: 8.50335257006105
Sum of squares total: 8.797537281099242

```

- In lines 2 to 5 we set the seed, and generate three different sized arrays of data, `x1Dat`, `x2Dat`, and `x3Dat`. These correspond to three treatments.
- In line 6 we create the array of arrays, `allData`.
- In line 7 the mean of each array is calculated via the `mean()` function. Note the use of the dot, or broadcast operator `'.'`, which performs the operation over each element in `allData`.
- In line 8 we calculate the length of each array via the `length()` function. Note that as above, the function is broadcast via `.` to all elements of `allData`. The resulting array of lengths is stored as `nArray`.

- In line 9 the point estimate for the population mean is calculated and stored as `xBarTotal`. Note that unlike in line 7, here we first vertically concatenate all the groups into a single array. This is done via the `vcat()` function, and the splat operator `...`, which performs the operation `vcat()` sequentially to each element of `allData`.
- In line 10, the total amount of groups is stored as `L`.
- In line 12, $SS_{Treatment}$ is calculated via (7.30). A comprehension is used, and the point estimate of the population mean `xbarTotal` is subtracted from the i th element of each array, and the results squared. These are each multiplied by the length of their respective arrays, and the results for each of the arrays summed together, and stored as `ssBetween` (note that “between” is sometimes used as an alternative name to “treatments”).
- In line 13, SS_{Error} is calculated via (7.30). The inner comprehension is used to square the difference between each observation, `ob`, and the group mean `xBarArray[i]`. The outer comprehension is used to repeat this process from the $1:L$ th group. The results for all groups are summed.
- In line 14, SS_{Total} is calculated via (7.27). The difference between each observation, `ob`, and the point estimate for the population mean, `xBarTotal`, is calculated and each result squared. This is first performed for the i th array, in the inner comprehension, and then repeated for all arrays via the outer comprehension. Finally all the squares are summed, via the outer `sum()` function.
- In lines 16 to 18 the sums of squares are printed. The results validate that (7.29) holds.

Carrying out ANOVA

Having understood the sum of squares decomposition we now present the F-statistic of ANOVA:

$$F = \frac{SS_{Treatment}/(L - 1)}{SS_{Error}/(m - L)}. \quad (7.31)$$

It is a ratio of the two sum of squares components of (7.29) normalized by their respective *degrees of freedom*, $L - 1$ and $m - L$. These normalized quantities are denoted by $MS_{Treatment}$ and MS_{Error} and hence $F = MS_{Treatment}/MS_{Error}$.

Under H_0 and with the model assumptions presented in (7.26), the ratio F follows an F -distribution (first introduced in Section 5.2) with $(L - 1, m - L)$ degrees of freedom. Intuitively, under H_0 we expect the numerator and denominator to have similar values, and hence expect F to be around 1 (indeed most of the mass of F distributions is concentrated around 1). However, if $MS_{Treatment}$ is significantly larger, then it indicates that H_0 may not hold. Hence the approach of the F -test is to reject H_0 if the F -statistic is greater than the $1 - \alpha$ quantile of the respective F -distribution. Similarly, the p -value for an observed F statistic f_o is given by,

$$p = \mathbb{P}(F_{L-1,m-L} > f_o).$$

where $F_{L-1,m-L}$ is an F -distributed random variable with $L - 1$ numerator degrees of freedom and $m - L$ denominator degrees of freedom.

It is often customary to summarize both the intermediate and final results of an ANOVA F-test in an *ANOVA table* as shown in Table 7.1 below, where “ T ” and “ E ” are shorthand for “Treatments” and “Error” respectively.

Source of variance:	DOF:	Sum of sq's:	Mean sum of sq's:	F-value:
Treatments (between treatments)	$L - 1$	SS_T	$MS_T = \frac{SS_T}{L - 1}$	$\frac{MS_T}{MS_E}$
Error (within treatments)	$m - L$	SS_E	$MS_E = \frac{SS_E}{m - L}$	
Total	$m - 1$	SS_{Total}		

Table 7.1: A one-way ANOVA table.

We now return to the three machines example and carry out a one-way ANOVA F -test. This is carried out in Listing 7.11 where we implement two alternative functions for ANOVA (at the time of writing of this edition Julia does not have a stand-alone ANOVA implementation). The first function, `manualANOVA()`, extends the sum of squares code presented in Listing 7.10 above. The second function, `glmANOVA()`, utilizes the GLM package that is described in detail in Chapter 8. Note that GLM requires the `DataFrames` package. Both implementations yield identical results, returning a tuple of the F -statistic and the associated p -value. In this example, the p -value is very small and hence under any reasonable α we would reject H_0 and conclude that there is sufficient evidence that the diameter of the bolt depends on the type of machine used.

Listing 7.11: Executing one way ANOVA

```

1  using GLM, Distributions, DataFrames
2
3  data1 = parse.(Float64, readlines("machine1.csv"))
4  data2 = parse.(Float64, readlines("machine2.csv"))
5  data3 = parse.(Float64, readlines("machine3.csv"))
6
7  function manualANOVA(allData)
8      nArray = length.(allData)
9      d = length(nArray)
10
11     xBarTotal = mean(vcat(allData...))
12     xBarArray = mean.(allData)
13
14     ssBetween = sum( [nArray[i]*(xBarArray[i] - xBarTotal)^2 for i in 1:d] )
15     ssWithin = sum([sum([(ob - xBarArray[i])^2 for ob in allData[i]])  

16                           for i in 1:d])
17     dfBetween = d-1
18     dfError = sum(nArray)-d
19
20     msBetween = ssBetween/dfBetween
21     msError = ssWithin/dfError
22     fStat = msBetween/msError
23     pval = ccdf(FDist(dfBetween,dfError),fStat)
24     return (fStat,pval)
25
26

```

```

27  function glmANOVA(allData)
28      nArray = length.(allData)
29      d = length(nArray)
30
31      treatment = vcat([fill(k,nArray[k]) for k in 1:d]...)
32      response = vcat(allData...)
33      dataFrame = DataFrame(Response=response, Treatment=categorical(treatment))
34      modelH0 = lm(@formula(Response ~ 1), dataFrame)
35      modelH1a = lm(@formula(Response ~ 1 + Treatment), dataFrame)
36      res = ftest(modelH1a.model, modelH0.model)
37      (res.fstat[1],res.pval[1])
38  end
39
40  println("Manual ANOVA: ", manualANOVA([data1, data2, data3]))
41  println("GLM ANOVA: ", glmANOVA([data1, data2, data3]))

```

```

Manual ANOVA: (10.516968568709117, 0.00014236168817139249)
GLM ANOVA: (10.516968568708988, 0.0001)

```

- In lines 7 to 25 the function `manualANOVA()` is implemented, which calculates the sum of squares in the same manner as in Listing 7.10. The sums of squares are normalized by their corresponding degrees of freedom `dfBetween` and `dfError`, and then in line 22 the F -statistic `fStat` is calculated. The p -value is then calculated in line 23 via the `ccdf()` function and the F -distribution `FDist()` with the degrees of freedom calculated above. The function then returns a tuple of values, comprising the F -statistic and the corresponding p -value.
- In lines 27 to 39 the function `glmANOVA()` is defined. This function calculates the F -statistic and p -value, via functionality in the GLM package, which is heavily discussed in Chapter 8. In lines 31 to 34 a Data-Frame (see Chapter 4) is set-up in the manner required by the GLM package. Then in lines 35 to 36 two ‘model objects’ are created via the `lm()` function from the GLM package. Note that `modelH0` is constructed on the assumption that the machine type has no effect on the response, while `modelH1` is constructed on the assumption that treatment has an effect. Finally, the `ftest()` function from the GLM package is used to compare if `modelH1a` fits the data ‘better’ than `modelH0`. Also note that the `model` fields of the model objects are used. Finally, the F -statistic and p -value are returned in line 38.
- The results of both functions are printed in lines 41 and 42, and it can be observed that the F -statistics and p -values calculated are identical to within the numerical error expected due to the different implementations.

More on the Distribution of the F -Statistic

Having explored the basics of ANOVA, we now use Monte Carlo simulation to illustrate that under H_0 the F -statistic is indeed distributed according to the F -distribution. In Listing 7.12 below, we present an example where Monte Carlo simulation is used to empirically generate the distribution of the F -statistic for two different cases where the number of groups is $L = 5$. In the first case, the means of each group are all the same (13.4), but in the second case, the means are

all different. The first case represents H_0 , while the latter does not. For both cases the standard deviation of each group is identical (2).

In this example, for each of the two cases, N sample runs are generated, where each run consists of a separate random collection of sample observations for each group. Hence by using a large number of sample runs N , histograms can be used to empirically represent the theoretical distributions of the F -statistics for both cases. The results show that the distribution of the F -statistics for the equal group means case is in agreement with the analytically expected F -distribution, while the F -statistic for the case of unequal group means is not.

Listing 7.12: Monte Carlo based distributions of the ANOVA F-statistic

```

1  using Random, Distributions, PyPlot
2  Random.seed! (808)
3
4  function anovaFStat(allData)
5      xBarArray = mean.(allData)
6      nArray = length.(allData)
7      xBarTotal = mean(vcat(allData...))
8      d = length(nArray)
9
10     ssBetween = sum( [nArray[i]*(xBarArray[i] - xBarTotal)^2 for i in 1:d] )
11     ssWithin = sum([sum([(ob - xBarArray[i])^2 for ob in allData[i]])
12                     for i in 1:d])
13     return (ssBetween/(d-1))/(ssWithin/(sum(nArray)-d))
14 end
15
16 case1 = [13.4, 13.4, 13.4, 13.4, 13.4]
17 case2 = [12.7, 11.8, 13.4, 11.7, 12.9]
18 stdDevs = [2, 2, 2, 2, 2]
19 numObs = [24, 15, 13, 23, 9]
20 L = length(case1)
21
22 N = 10^5
23
24 mcFstatsH0 = Array{Float64}(undef, N)
25 for i in 1:N
26     mcFstatsH0[i] = anovaFStat([ rand(Normal(case1[j],stdDevs[j]),numObs[j])
27                                     for j in 1:L ])
28 end
29
30 mcFstatsH1 = Array{Float64}(undef, N)
31 for i in 1:N
32     mcFstatsH1[i] = anovaFStat([ rand(Normal(case2[j],stdDevs[j]),numObs[j])
33                                     for j in 1:L ])
34 end
35
36 plt[:hist](mcFstatsH0, 100, color="b", histtype="step",
37             normed="true", label="Equal group \nmeans case")
38 plt[:hist](mcFstatsH1, 100, color="r", histtype="step",
39             normed="true", label="Unequal group \nmeans case")
40
41 dfBetween = L - 1
42 dfError = sum(numObs) - 1
43 xGrid = 0:0.01:10
44 plot(xGrid, pdf(FDist(dfBetween, dfError),xGrid),
45       color="b", label="f-statistic \nanalytic")

```

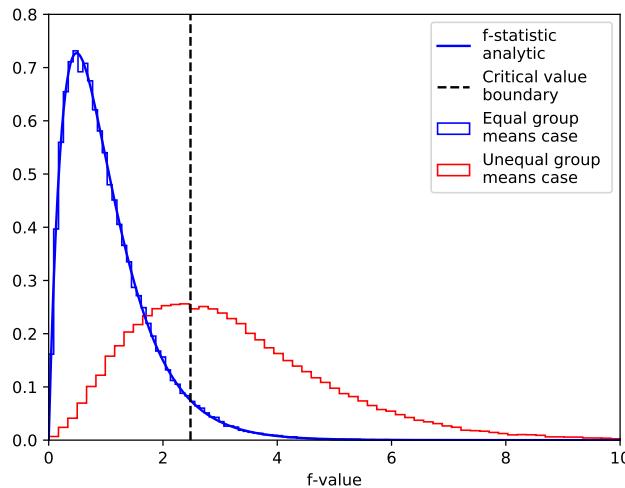


Figure 7.5: Histograms of the f-statistic for the case of equal group means, and unequal group means, along with the analytic PDF of the f distribution.

```

46 critVal = quantile(FDist(dfBetween, dfError), 0.95)
47 plot([critVal, critVal], [0,0.8], "k--", label="Critical value \nboundary")
48 xlim(0,10)
49 ylim(0,0.8)
50 xlabel("f-value")
51 legend(loc="upper right")

```

- In lines 4 to 14 we create the function `anovaFStat()`, which takes an array of arrays as input, calculates the sums of squares and mean sums of squares as per Table 7.1, and returns the F -statistic of the data. Note that lines 5 to 11 are the same as lines 7 to 13 from Listing 7.10.
- In lines 16 and 17, we create two arrays for our two cases. `case1`, represents an array of means for the case of all means being equal, and `case2`, represents an array of means for the case of all means not equal equal (note the i^{th} element of each array is the mean of the i^{th} group, or level, 1 to 5).
- In line 18 we create the array of group standard deviations, `stdDevs`. Note that in both cases of equal group means and unequal group means, the standard deviations of all the groups are equal (i.e. 2).
- In line 19 we create the array `numObs`, where each element represents the number of observations of the i^{th} group, or level.
- In line 20 the total number of groups is stored as `L`.
- In line 22 we specify the total number of Monte Carlo runs to be performed, `N`.
- In line 24 we preallocate the array `mcFStatsH0`, which will store `N` Monte Carlo generated f-statistics, for the case of all group means equal.

- In lines 25 to 28, we use a loop to generate N F -statistics via the `anovaFStat()` function defined earlier. We first use the `rand()` and `Normal()` functions within a comprehension to generate data for each of the sample groups, using the group means, standard deviations, and number of observations as specified in the arrays `meansH0`, `stdDevs` and `numObs` respectively. The comprehension generates an array of arrays, where each of the five elements of the outermost array is another array containing the observations for that group, 1 to 5. This array of arrays is then used as the argument for `anovaFStat()`, which carries out a one-way ANOVA test on the data and outputs the corresponding F -value. This whole process is repeated N times via the outermost `for` loop. Hence `mcfstatsH0` is populated with N Monte Carlo generated f -statistics for the case of group means being equal.
- Lines 30 to 34 essentially repeat the steps of lines 24 to 28, except for `case2`, where the means of each group are not equal. The resulting N F -statistics calculated are stored in the pre-allocated array `mcfstatH1`.
- In lines 36 to 39 histograms of the f -statistics calculated for our two cases are plotted.
- In lines 41 and 42 the degrees of freedom of the treatments `dfBetween`, and the degrees of freedom of the error `dfError` are calculated.
- In lines 44 and 45 the analytic PDF of the F -statistic for our example is plotted. The `pdf()` function along with the `FDist()` function from the `Distributions` package are used, and the degrees of freedom calculated in lines 41 and 42 used as inputs.
- In line 46 the `quantile()` function is used to calculate the 95^{th} quantile of the f -distribution, `FDist()`, given the degrees of freedom of our problem for the given significance level $\alpha = 0.05$. This value represents the boundary of the rejection region, which is given by the area to the right (and is 5% of the total area under the PDF).
- In line 47, the critical value boundary is plotted.

Extensions

In this section we have only touched on the very basics of ANOVA, and elaborated it through examples around the one-way ANOVA case. This stands at the basis of *experimental design*. However, there are many more aspects to ANOVA, and related ideas that one can explore. These include, but are not limited to:

- Extensions to *two-way ANOVA* where there are two treatment categories, for example “machine type” and “type of lubricant used in the machine”.
- Higher dimensional extensions, which are often considered in *block factorial design*.
- Comparison of individual factors to determine which specific treatments have an effect and in which way.
- Aspects of optimal experimental design.

These, and many more aspects can be found in design and analysis of experiment texts, such as for example [Mon17]. At the time of writing, many such procedures are not implemented directly in Julia. However, one alternative is the *R* software package, which contains many different implementations of these ANOVA extensions, among others. One can call these *R* packages directly from Julia.

7.4 Independence and Goodness of Fit

We now consider a different group of hypothesis tests and associated procedures that deal with *checking for independence* and more generally checking *goodness of fit*. One question often posed is: “Does the population follow a specific distributional form?”. We may hypothesize that the distribution is normal, exponential, Poisson, or that it follows any other form (see Chapter 3 for an extensive survey of probability distributions). Checking such a hypothesis is loosely called goodness of fit. Furthermore, in the case of observations over multiple dimensions, we may hypothesize that the different dimensions are independent. Checking for such independence is similar to the goodness of fit check.

In order to test for goodness of fit against some hypothesized distribution F_0 , we setup the hypothesis test as,

$$H_0 : X \sim F_0, \quad \text{vs.} \quad H_1 : \text{otherwise.} \quad (7.32)$$

Here X denotes an arbitrary random variable from the population. In this case, we consider the parameter space associated with the test as the space of all probability distributions. The hypothesis formulation then partitions this space into $\{F_0\}$ (for H_0) and all other distributions in H_1 .

For the independence case, assume X is a vector of two random variables, say $X = (X_1, X_2)$. Then for this case the hypothesis test setup would be,

$$H_0 : X_1 \text{ independent of } X_2 \quad \text{vs.} \quad H_1 : X_1 \text{ not independent of } X_2. \quad (7.33)$$

This sets the space of H_0 as the space of all distributions of independent random variable pairs, and H_1 as the complement.

To handle hypotheses such as (7.32) and (7.33) we introduce two different test procedures, the *Chi-squared test* and the *Kolmogorov-Smirnov test*. The Chi-squared test is used for goodness of fit of discrete distributions and for checking independence, while the Kolmogorov-Smirnov test is used for goodness of fit for arbitrary distributions based on the empirical cumulative distribution function. Before we dive into the individual test examples, we explain how to construct the corresponding test statistics.

In the Chi-squared case, the approach involves looking at counts of observations that match disjoint categories $i = 1, \dots, M$. For each category i , we denote O_i as the number of observations that match that category. In addition, for each category there is also an expected number of observations under H_0 , which we denote as E_i . With these, one can express the test statistic as,

$$\chi^2 = \sum_{i=1}^M \frac{(O_i - E_i)^2}{E_i}. \quad (7.34)$$

Notice that under H_0 of (7.32), we expect that for each category i , both O_i and E_i will be relatively close, and hence it is expected that the sum of relative squared differences, χ^2 , will not be too big. Conversely, a large value of χ^2 may indicate that H_0 is not plausible. Later in this section, we show how to construct the test to check for both goodness of fit (7.32), and to check for independence (7.33).

In the case of Kolmogorov-Smirnov, a key aspect is the empirical cumulative distribution function (ECDF), which was introduced in Section 4.5. Recall that for a sample of observations, x_1, \dots, x_n the ECDF is,

$$\hat{F}(x) = \frac{1}{n} \sum_{i=1}^n \mathbf{1}\{x_i \leq x\} \quad \text{where } \mathbf{1}\{\cdot\} \text{ is the indicator function.} \quad (7.35)$$

The approach of Kolmogorov-Smirnov test is to check the closeness of the ECDF to the CDF hypothesized under H_0 in (7.32). This is done via the *Kolmogorov-Smirnov statistic*,

$$\tilde{S} = \sup_x |\hat{F}(x) - F_0(x)|, \quad (7.36)$$

where $F_0(\cdot)$ is the CDF under H_0 and sup is the supremum over all possible x values. Similar to the case of Chi-squared, under H_0 it is expected that $\hat{F}(\cdot)$ does not deviate greatly from $F_0(\cdot)$, and hence it is expected that \tilde{S} is not very large.

The key to both the Chi-Squared and Kolmogorov-Smirnov tests is that under H_0 there are tractable known approximations to the distribution of the test statistics of both (7.34) and (7.36)). These approximations allow us to obtain an approximate p -value in the standard way via,

$$p = \mathbb{P}(W > u), \quad (7.37)$$

where W denotes a random variable distributed according to the approximate distribution and u is the observed test statistic of either (7.34) or (7.36). We now elaborate on the details.

Chi-squared Test for Goodness of Fit

Consider the hypothesis (7.32) and assume that the distribution F_0 can be partitioned into categories $i = 1, \dots, M$. Such a partition naturally occurs when the distribution is discrete with a finite number of outcomes. It can also be artificially introduced in other cases. With such a partition, having n sample observations, we denote by E_i the expected number of observations satisfying category i . These values are theoretically computed. Then, based on observations x_1, \dots, x_n , we denote by O_i as the number of observations that satisfy category i . Note that,

$$\sum_{i=1}^M E_i = n, \quad \text{and} \quad \sum_{i=1}^M O_i = n.$$

Now, based on $\{E_i\}$ and $\{O_i\}$, we can compute the χ^2 test statistic (7.34).

It turns out that under H_0 , the χ^2 test statistic of (7.34) approximately follows a Chi-squared distribution with $M - 1$ degrees of freedom. Hence this allows us to approximate the p -value via

(7.37), where W is taken as such a Chi-squared random variable and u as the test statistic. This is also sometimes called *Pearson's chi-squared test*.

We now present an example where we assume under H_0 that a die is biased, with the probabilities for each side (1 to 6) given by the following vector \mathbf{p} ,

$$\mathbf{p} = (0.08, \quad 0.12, \quad 0.2, \quad 0.2, \quad 0.15, \quad 0.25).$$

Note that if there are then n observations, we have that $E_i = n p_i$. For this example $n = 60$, and hence the vector of expected values for each side is,

$$\mathbf{E} = (4.8, \quad 7.2, \quad 12, \quad 12, \quad 9, \quad 15).$$

Now imagine that the die is rolled $n = 60$ times, and the following count of outcomes (1 to 6)) observed,

$$\mathbf{O} = (3, \quad 2, \quad 9, \quad 11, \quad 8, \quad 27).$$

In Listing 7.13 below, we use this data to compute the test statistic and p -value. This is done first manually, and then the `ChisqTest()` function from the `HypothesisTests` package is used.

Listing 7.13: Chi-squared test for goodness of fit

```

1  using Distributions, HypothesisTests
2
3  p = [0.08, 0.12, 0.2, 0.2, 0.15, 0.25]
4  O = [3, 2, 9, 11, 8, 27]
5  M = length(O)
6  n = sum(O)
7  E = n*p
8
9  testStatistic = sum((O-E).^2 ./E)
10 pVal = ccdf(Chisq(M-1), testStatistic)
11
12 println("Manually calculated test statistic: ", testStatistic)
13 println("Manually calculated p-value: ", pVal, "\n")
14
15 println(ChisqTest(O,p))

```

```

Manually calculated test statistic: 14.97499999999998
Manually calculated p-value: 0.010469694843220351

```

```

Pearson's Chi-square Test
-----
Population details:
    parameter of interest: Multinomial Probabilities
    value under h_0:          [0.08, 0.12, 0.2, 0.2, 0.15, 0.25]
    point estimate:           [0.05, 0.0333333, 0.15, 0.183333, 0.133333, 0.45]
    95% confidence interval: Tuple{Float64,Float64}[(0.0, 0.1828), (0.0, 0.1662),
                                                (0.0333, 0.2828), (0.0667, 0.3162), (0.0167, 0.2662), (0.3333, 0.5828)]

```

```

Test summary:
    outcome with 95% confidence: reject h_0
    one-sided p-value:           0.0105

```

```

Details:
    Sample size:               60

```

```

statistic:          14.975000000000001
degrees of freedom: 5
residuals:          [-0.821584, -1.93793, -0.866025, -0.288675, -0.333333, 3.09839]
std. residuals:     [-0.85656, -2.06584, -0.968246, -0.322749, -0.361551, 3.57771]

```

- In line 3 the array p is created, which represents the probabilities of each side occurring under H_0 .
- In line 4 the array O is created, which contains the frequencies, or counts, of each side outcome observed.
- In line 5 the total number of categories (or side outcomes) is stored as M .
- In line 6 the total number of observations is stored as n .
- In line 7, the array of expected number of observed outcomes for each side is calculated by multiplying the vector of expected probabilities under H_0 by the total number of observations n . The resulting array is stored as E .
- In line 9 the equation (7.34) is used to calculate the Chi-squared test statistic.
- In line 10 the test statistic is used to calculate the p -value. Since under the null hypothesis the test statistic is asymptotically distributed according to a chi-squared distribution, the `ccdf()` function is used on a `Chisq()` distribution with $M-1$ degrees of freedom.
- In lines 12 and 13 the manually calculated test statistic and p -value are printed.
- In line 15 the `ChisqTest()` function from the `HypothesisTests` package is used to perform the chi-squared test on the frequency data in array p . It can be observed that the test statistic and p -values match those calculated manually. In this case, the null hypothesis is rejected at the 5% significance level (i.e. there is sufficient evidence to believe the die is weighted differently to the weights in p).

Chi-squared Test Used to Check Independence

We now show how a Chi-squared statistic can be used to check for independence, as in (7.33). Consider the following example, where 373 individuals are categorized as Male/Female, and Smoker/Non-smoker, as in the following *contingency table*,

	Smoker	Non-smoker
Male	18	132
Female	45	178

In this example, 18 individuals were recorded as “male” and “smoker”, and so forth. Now, under H_0 , we assume that the smoking or non-smoking behavior of the individual is independent of the gender (male or female). To check for this using a Chi-squared statistic, we first setup $\{E_i\}$ and $\{O_i\}$ as in the following table,

	Smoker	Non-smoker	Total/proportion
Male	$O_{11} = 18$ $E_{11} = 25.34$	$O_{12} = 132$ $E_{12} = 124.67$	150 / 0.402
Female	$O_{21} = 45$ $E_{21} = 37.66$	$O_{22} = 178$ $E_{22} = 185.33$	223 / 0.598
Total/Proportion	63/0.169	310/0.831	373 / 1

Table 7.2: The elements $\{O_{ij}\}$ and $\{E_{ij}\}$ as in a Contingency table.

Here the observed *marginal distribution* over male vs. female is based on the proportions $\mathbf{p} = (0.402, 0.598)$ and the distribution over smoking vs. non-smoking is based on the proportions $\mathbf{q} = (0.169, 0.831)$. Then, since independence is assumed under H_0 , we multiply the marginal probabilities to obtain the expected observation counts,

$$E_{ij} = n p_i q_j, \quad (7.38)$$

For example, $E_{21} = 373 \times 0.169 \times 0.598 = 37.66$. Now with these values at hand, the Chi-squared test statistic can be setup as follows,

$$\chi^2 = \sum_{i=1}^m \sum_{j=1}^{\ell} \frac{(O_{ij} - E_{ij})^2}{E_{ij}}, \quad (7.39)$$

where m and ℓ are the respective dimensions of the contingency table ($m = \ell = 2$ in this example).

It turns out that under H_0 , this statistic is approximately Chi-squared distributed, with $(m-1)(\ell-1)$ degrees of freedom (1 degree of freedom in this case). Hence (7.37) can then be used to determine an (approximate) p -value for this test, just like in the previous example.

Listing 7.14 below carries out a chi-squared test in order to check if there is a relationship between gender and smoking. In this example, since the p -value is 0.58, we conclude by saying there is insufficient evidence that there is a relationship (i.e. insufficient evidence to reject H_0) under any sensible significance level.

Listing 7.14: Chi-squared for checking independence

```

1  using Distributions
2
3  xObs      = [18 132; 45 178]
4  rowSums   = [sum(xObs[i,:]) for i in 1:2]
5  colSums   = [sum(xObs[:,i]) for i in 1:2]
6  n         = sum(xObs)
7
8  rowProps = rowSums/n
9  colProps = colSums/n
10
11 xExpect  = [colProps[c]*rowProps[r]*n for r in 1:2, c in 1:2]
12
13 testStat = sum([(xObs[r,c]-xExpect[r,c])^2 / xExpect[r,c] for r in 1:2,c in 1:2])
14

```

```

15 pVal = ccdf(Chisq(1),testStat)
16
17 println("Chi-squared value: ", testStat)
18 println("P-value: ", pVal)

```

```

Chi-squared value: 4.274080056208799
P-value: 0.03869790606536347

```

- In line 3 the observations in the contingency table is stored as the 2-dimensional array `xObs`.
- In line 4 the observations in each row are summed via `xObs[i, :]`, and the use of a comprehension.
- In line 5 the observations in each column are calculated via a similar approach to that in line 4 above.
- In line 6 the total number of observations is stored as `n`.
- In line 8 and 9 the row and column proportions are calculated.
- In line 11 the expected number of observations, $\{E_{ij}\}$ (shown in Table 7.2), are calculated. Note the use of the comprehension, which calculates (7.38) for each combination of sex and smoker/non-smoker.
- In line 13 the test statistic is calculated via (7.39) through the use of a comprehension.
- In line 15 the test statistic is used to calculate the p -value. Since under the null hypothesis the test statistic is asymptotically distributed according to a chi-squared distribution, the `ccdf()` function is used on a `Chisq()` distribution with $(m - 1)(\ell - 1)$ degrees of freedom (i.e. 1 in this example).
- In lines 17 and 18 the test statistic and corresponding p -value are printed, and since $p = 0.039$, we conclude by stating there is sufficient evidence to reject H_0 .

Kolmogorov-Smirnov Test

We now depart from the situations of a finite number of categories as in the Chi-squared test, and consider the Kolmogorov-Smirnov test, which is based on the test statistic (7.36). The approach is based on the fact that, under H_0 of (7.32), the empirical cumulative distribution (ECDF) $\hat{F}(\cdot)$ is close to the actual CDF $F_0(\cdot)$. To get a feel for this notice that for every value $x \in \mathbb{R}$, the ECDF at that value, $\hat{F}(x)$, is the proportion of the number of observations less than or equal to x . Under H_0 , multiplying the ECDF by n yields a binomial random variable with success probability, $F_0(x)$:

$$n \hat{F}(x) \sim \text{Bin}(n, F_0(x)).$$

Hence,

$$\mathbb{E}[\hat{F}(x)] = F_0(x), \quad \text{Var}(\hat{F}(x)) = \frac{F_0(x)(1 - F_0(x))}{n}.$$

See the Binomial distribution in Section 3.5. Hence, for non-small n , the ECDF and CDF should be close since the variance for every value x is of the order of $1/n$ and hence diminishes as n grows. The formal statement of this is known as the *Glivenko Cantelli Theorem*.

For finite n , the ECDF will not exactly align with the CDF. However the Kolmogorov-Smirnov test statistic (7.36) is useful when it comes to measuring this deviation. This is because the *stochastic process* in the variable, x ,

$$\sqrt{n}(\hat{F}(x) - F_0(x)) \quad (7.40)$$

is approximately identical in probability law to a standard *Brownian Bridge*, $B(\cdot)$, composed with $F_0(x)$. That is, by denoting $\hat{F}_n(\cdot)$ as the ECDF with n observations, we have that

$$\sqrt{n}(\hat{F}_n(x) - F_0(x)) \approx^d B(F_0(x)),$$

which asymptotically converges to equality in distribution as $n \rightarrow \infty$. Note that a Brownian Bridge, $B(t)$, is a form of a variant of *Brownian Motion*, constrained to equal 0 both at $t = 0$ and $t = 1$. It is a type of *diffusion process*.

Now consider the supremum as in the Kolmogorov-Smirnov test statistic, \tilde{S} as defined in (7.36). It can be shown that, in cases where $F_0(\cdot)$ is continuous, as $n \rightarrow \infty$,

$$\sqrt{n}\tilde{S} \stackrel{d}{=} \sup_{t \in [0,1]} |B(t)|. \quad (7.41)$$

Importantly, notice that the right hand side does not depend on $F_0(\cdot)$, but rather is the maximal value attained by the absolute value of the Brownian bridge process over the interval $[0, 1]$. It then turns out that (see for example [Man07] for a derivation) such a random variable, denoted by K , has CDF,

$$F_K(x) = \mathbb{P}\left(\sup_{t \in [0,1]} |B(t)| \leq x\right) = 1 - 2 \sum_{k=1}^{\infty} (-1)^{k-1} e^{-2k^2 x^2} = \frac{\sqrt{2\pi}}{x} \sum_{k=1}^{\infty} e^{-(2k-1)^2 \pi^2 / (8x^2)}. \quad (7.42)$$

This is sometimes called the *Kolmogorov distribution*. Thus to obtain a p -value for the Kolmogorov-Smirnov test using (7.37) we calculate,

$$p = 1 - F_K(\sqrt{n}\hat{S}). \quad (7.43)$$

In Listing 7.15 below, we show a comparison of the distributions of the K-S test statistic multiplied by a factor of \sqrt{n} as on the left hand side of (7.41), and the distribution of the random variable, K , as on the right hand side of (7.41). This is done for two different scenarios. In the first, the test statistics are calculated based on data sampled from an exponential distribution, and in the second, the test statistics are calculated based on data sampled from a normal distribution. As illustrated in the resulting Figure 7.6, the distributions of the Monte Carlo generated test statistics are in agreement with the analytic PDF, regardless of what underlying distribution $F_0(x)$ the data comes from.

Listing 7.15: Comparisons of distributions of the K-S test statistic

```

1  using Random, Distributions, StatsBase, HypothesisTests, PyPlot
2  Random.seed!(0)
3

```

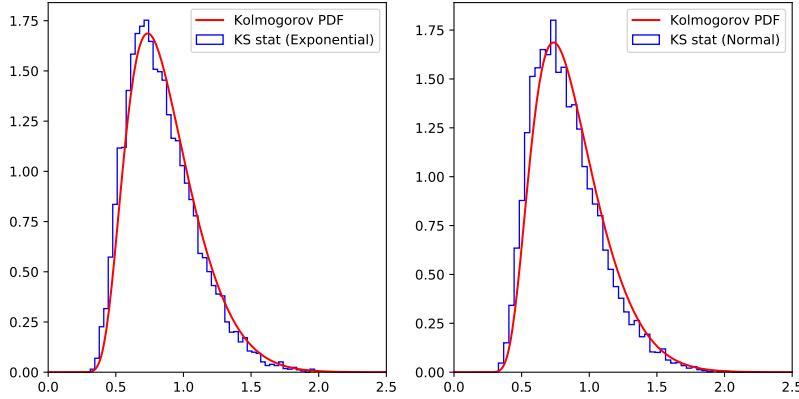


Figure 7.6: PDF of the Kolmogorov distribution, alongside histograms of K-S test statistics from normal and exponential populations.

```

4  function ksStat(dist)
5      data = rand(dist,n)
6      Fhat = ecdf(data)
7      sqrt(n)*maximum(abs.(Fhat(xGrid) - cdf(dist,xGrid)))
8  end
9
10 n = 25
11 N = 10^5
12 xGrid = -10:0.001:10
13 kGrid = 0:0.01:5
14
15 dist1 = Exponential(1)
16 kStats1 = [ksStat(dist1) for _ in 1:N]
17
18 dist2 = Normal()
19 kStats2 = [ksStat(dist2) for _ in 1:N]
20
21 figure(figsize=(10,5))
22 subplot(121)
23 plt[:hist](kStats1,50,color="b",label="KS stat (Exponential)",histtype="step",
24             normed=true)
25 plot(kGrid,pdf(Kolmogorov(),kGrid),"r",label="Kolmogorov PDF")
26 legend(loc="upper right")
27 xlim(0,2.5)
28
29 subplot(122)
30 plt[:hist](kStats2,50,color="b",label="KS stat (Normal)",histtype="step",
31             normed=true)
32 plot(kGrid,pdf(Kolmogorov(),kGrid),"r",label="Kolmogorov PDF")
33 legend(loc="upper right")
34 xlim(0,2.5)

```

- In lines 4 to 8, the function `ksStat()` is created, which takes a distribution type as input, randomly samples `n` observations from it, calculates the ECDF of the data via the `ecdf()` function, and finally returns the left hand side of (7.41) by calculating the K-S test statistic via (7.36), and multiplying this by `sqrt(n)`. Note that in line 6, the `ecdf()` function returns

a cdf function type itself, which is stored as `Fhat`, and evaluated over `xGrid` in line 7.

- In line 10 the total number of observations that each sample group will contain is specified as `n`.
- In line 11 the total number of sample groups is specified as `N`.
- In line 12 we specify the domain over which `ksStat` will be calculated, `xGrid`. Note that the increments must be small enough to capture every piecewise difference between the CDF and ECDF, and must cover the domain between their start and end points. Hence the nominally chosen range and step size of `xGrid`.
- In line 13 the grid over which the KS test statistics are plotted is specified as `kGrid`.
- In line 15 a exponential distribution object, with a mean of 1, is created and stored as `dist1`.
- In line 16, a comprehension is used along with the `ksStat()` function to generate `N` K-S test statistics, based on sample groups where the data has been randomly sampled from the exponential distribution `dist1`. The test statistics are stored in the array `kStats1`.
- In line 18 a standard normal distribution object is created and stored as `dist2`.
- In line 19, `N` K-S test statistics are calculated based on random samples drawn from `dist2`, in a similar manner to that in line 16.
- In lines 21 to 34 Figure 7.6 is created. In lines 23 to 25, a histogram of the K-S test statistics stored in `kStats1` are plotted alongside the analytic PDF of the Kolmogorov distribution, while in lines 30 to 32 the same process is repeated for the test statistics stored in `kStats2`. Note the use of the `Kolmogorov()` function from the `Distributions` package on lines 25 and 32.
- From the results of Figure 7.6, it can be seen that regardless of the type of underlying distribution from which the data comes from, $F(x)$, the distribution of the Monte Carlo generated K-S test statistics agree with the analytic PDF of the Kolmogorov distribution.

Now that we have demonstrated that the distribution of the scaled Kolmogorov-Smirnov statistic is similar to the distribution of K as in (7.42), we demonstrate how the Kolmogorov-Smirnov statistic can be used to carry out a goodness of fit test. For this example, consider that a series of observations have been made from some unknown underlying gamma distribution with shape parameter 2 and mean 5. The question we then wish to ask is: given the sample observations, is the underlying distribution exponential?

To help illustrate the logic of the approach, Listing 7.16 plots the ECDF $\hat{F}_0(x)$ against the true CDF $F_0(x)$ (note that in a realistic scenario, the true CDF will not be known). The stochastic process of (7.40) is also plotted with the horizontal axis rescaled. Now, since under the null hypothesis that the data comes from a specified distribution with CDF $F_0(x)$ the difference behaves like a Brownian bridge. It can be observed that, while the blue line appears to behave like a “typical” Brownian bridge, the red line does not. This suggests that the data does not come from the postulated exponential F_0 . The listing also calculates the p -value of the K-S test manually, as well as via `ApproximateOneSampleKSTest()` from the `HypothesisTests` package. The resulting p -values are in agreement.

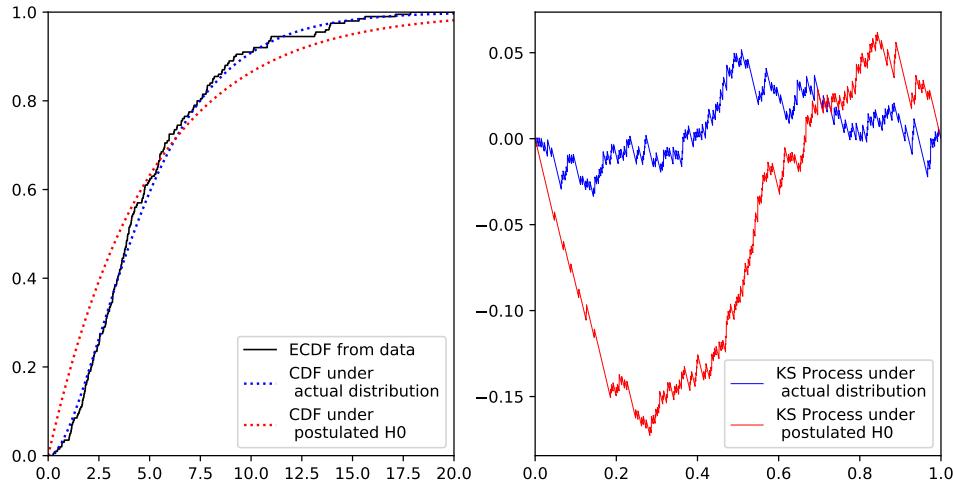


Figure 7.7: Left: CDFs and ECDF. Right: K-S processes scaled over $[0, 1]$.

Listing 7.16: ECDF, actual and postulated CDF's, and their differences

```

1  using Random, Distributions, StatsBase, PyPlot, HypothesisTests
2  Random.seed!(2)
3
4  dist = Gamma(2, 2.5)
5  distH0 = Exponential(5)
6  n = 200
7  data = rand(dist,n)
8  Fhat = ecdf(data)
9  diffF(dist, x) = Fhat(x) - cdf(dist,x)
10 xGrid = 0:0.001:30
11
12 figure(figsize=(10,5))
13 subplot(121)
14 plot(xGrid,Fhat(xGrid),"k",lw=1, label="ECDF from data")
15 plot(xGrid,cdf(dist,xGrid),"b:",label="CDF under \n actual distribution")
16 plot(xGrid,cdf(distH0,xGrid),"r:",label="CDF under \n postulated H0")
17 legend(loc="lower right")
18 xlim(0,20);ylim(0,1);
19
20 subplot(122)
21 plot(cdf(dist,xGrid), diffF(dist, xGrid),lw=0.5, "b",
22       label="KS Process under \n actual distribution")
23 plot(cdf(distH0,xGrid), diffF(distH0, xGrid),lw=0.5, "r",
24       label="KS Process under \n postulated H0")
25 legend(loc="lower right")
26 xlim(0,1)
27
28 N = 10^5
29 KScdf(x) = sqrt(2pi)/x*sum([exp(-(2k-1)^2*pi^2 ./ (8x.^2)) for k in 1:N])
30 ksStat = maximum(abs.(diffF(distH0, xGrid)))
31
32 println("p-value calculated via series: ",
33         1-KScdf(sqrt(n)*ksStat))
34 println("p-value calculated via Kolmogorov distribution: ",

```

```

35      1-cdf(Kolmogorov(),sqrt(n)*ksStat),"\n")
36
37  println(ApproximateOneSampleKSTest(data,distH0))

```

```

p-value calculated via series: 1.3257956569923124e-5
p-value calculated via Kolmogorov distribution: 1.3257956569923124e-5

Approximate one sample Kolmogorov-Smirnov test
-----
Population details:
  parameter of interest: Supremum of CDF differences
  value under h_0:          0.0
  point estimate:          0.17277717826644445

Test summary:
  outcome with 95% confidence: reject h_0
  two-sided p-value:           <1e-4

Details:
  number of observations:    200
  KS-statistic:              2.443438287729597

```

- In lines 4 and 5 the actual underlying distribution, and postulated distribution under H_0 , are defined as `dist` and `distH0` respectively.
- In lines 6 to 7 our sample data is generated, for $n=200$ observations.
- In line 8 the ECDF is defined as `Fhat` via the `ecdf()` function.
- In line 9 the function `diffF()` is created, which calculates the difference between the ECDF `Fhat` generated from the data, and the CDF of the postulated input distribution `dist` at the point `x`. This is the implementation of the inner component of the right hand side of (7.36).
- In line 10 the grid of values over which the CDF's and ECDF will be plotted is specified as `xGrid`.
- Lines 12 to 26 generate Figure 7.7. In line 14 to 16 the ECDF `Fhat`, the CDF of the actual (unknown) distribution `dist`, and the CDF of the postulated distribution under H_0 `distH0` are plotted.
- In lines 21 to 24 the KS process is plotted for both the true underlying case (`dist`), as well as for the postulated distribution under H_0 (`distH0`) are plotted. Note that the blue line (true underlying case) follows a “typical” Brownian bridge, while the red line (postulated case) does not.
- In lines 28 to 37 the p -value of the KS test statistic is calculated in three different ways. First, it is calculated via the implementation of (7.42), then via the use of the `Kolmogorov()` distribution, and finally via the use of the `ApproximateOneSampleKSTest()` function from the `Distributions` package.
- In line 28 the total number of KS test statistics to be calculated is specified as `N`.
- In line 29 (7.42) is implemented as the function `KScdf()`.

- In line 30 the KS test statistic is calculated through the implementation of equation (7.36). This is done via the combination of the previously defined `diffF()` function from line 9 with the `maximum()` and `abs()` functions.
- In lines 32 and 33, the p -value is calculated via (7.43), where the value of the CDF of the Kolmogorov distribution, given by `KScdf()`, is evaluated at the test statistic given by `sqrt(n)*ksStat`.
- In lines 34 and 35 the p -value is calculated in the same way as in lines 32 and 33, however in this case the CDF function `cdf()` is used on the `Kolmogorov()` distribution from the `Distributions` package. Note that the test statistic in this case is the same as in line 33.
- Finally, in line 37, the hypothesis test is performed via the `ApproximateOneSampleKSTest()` function from the `Distributions` package.
- It can be seen from the output that the three p -values calculated via the three different methods are all in agreement. The resulting small p -values indicate that there is statistical significance to reject H_0 , or in other words, there is statistical significance that the underlying data does not come from the postulated distribution `distH0`.

7.5 Power Curves

In this section, the concept of Power is covered in depth. Recall that, as first introduced in Section 5.6 and summarized in Table 5.1, the statistical *power* of a hypothesis test is the probability of correctly rejecting H_0 . We now reinforce this idea through the following introductory example. Consider a normal population, with unknown parameters μ and σ , and say that we wish to conduct a one sided hypothesis test on the population mean using the following hypothesis test set-up,

$$H_0 : \mu = \mu_0 \quad H_1 : \mu > \mu_0. \quad (7.44)$$

Importantly, since power is the probability of a correct rejection, if in conducting a hypothesis test, the underlying parameter varies greatly from the value under the null hypothesis, then the power of the test in this scenario is greater. Likewise, if the underlying parameter does not vary greatly from the value under the null hypothesis, the power of the test is less.

In Listing 7.17 below, several different scenarios are considered, and for each, N test statistics are calculated via Monte Carlo for N sample groups. In the first scenario, the underlying mean equals the mean under the null hypothesis, and in each subsequent scenario, the parameters are changed, such that the actual underlying mean deviates further and further from μ_0 . Kernel density estimation is then used on the test statistics for each scenario, and the resulting numerically estimated PDF's of the test statistics are plotted, along with the output of the numerically approximated power. In a way the resulting Figure 7.8 is similar to Figure 5.12, however in this case the focus is on power, which is given by, $1 - \mathbb{P}(\text{Type II error})$ (i.e. the probability of correctly rejecting H_0). The power of the hypothesis under different scenarios is given by the area under each PDF to the right of the critical value boundary. Hence, it can be observed that the larger the difference between the actual parameter and the value being tested under H_0 , the greater the power of the test.

Listing 7.17: Distributions under different hypotheses

```

1  using Random, Distributions, KernelDensity, PyPlot
2  Random.seed!(1)
3
4  function tStat(mu0,mu,sig,n)
5      sample = rand(Normal(mu,sig),n)
6      xBar    = mean(sample)
7      s       = std(sample)
8      (xBar-mu0) / (s/sqrt(n))
9  end
10
11 mu0, mu1A, mu1B = 20, 22, 24
12 sig, n = 7, 5
13 N = 10^6
14 alpha = 0.05
15
16 dataH0  = [tStat(mu0,mu0,sig,n) for _ in 1:N]
17 dataH1A = [tStat(mu0,mu1A,sig,n) for _ in 1:N]
18 dataH1B = [tStat(mu0,mu1B,sig,n) for _ in 1:N]
19 dataH1C = [tStat(mu0,mu1B,sig,2*n) for _ in 1:N]
20 dataH1D = [tStat(mu0,mu1B,sig/2,2*n) for _ in 1:N]
21
22 tCrit = quantile(TDist(n-1),1-alpha)
23 estPwr(sample) = sum(sample .> tCrit)/N
24
25 println("Rejection boundary: ", tCrit)
26 println("Power under H0: ", estPwr(dataH0))
27 println("Power under H1A: ", estPwr(dataH1A))
28 println("Power under H1B (mu's farther apart): ", estPwr(dataH1B))
29 println("Power under H1C (double sample size): ", estPwr(dataH1C))
30 println("Power under H1D (like H1C but std/2): ", estPwr(dataH1D))
31
32 kH0  = kde(dataH0)
33 kH1A = kde(dataH1A)
34 kH1D = kde(dataH1D)
35 xGrid = -10:0.1:15
36
37 plot(xGrid,pdf(kH0,xGrid),"b", label="Distribution under H0")
38 plot(xGrid,pdf(kH1A,xGrid),"r", label="Distribution under H1A")
39 plot(xGrid,pdf(kH1D,xGrid),"g", label="Distribution under H1D")
40 plot([tCrit,tCrit],[0,0.4],"k--", label="Critical value boundary")
41 xlim(-5,10)
42 ylim(0,0.4)
43 xlabel(L"\Delta = \mu - \mu_0")
44 legend(loc="upper right")

```

```

Rejection boundary: 2.131846786326649
Power under H0: 0.049598
Power under H1A: 0.134274
Power under H1B (mu's farther apart): 0.281904
Power under H1C (double sample size): 0.406385
Power under H1D (like H1C but std/2): 0.91554

```

- In lines 4 to 9 the function `tStat` is defined, which returns the value of the test statistic for a randomly generated group of sample observations. It does this by generating `n` sample observations from an underlying gaussian process given the input arguments `mu` and `sig` in

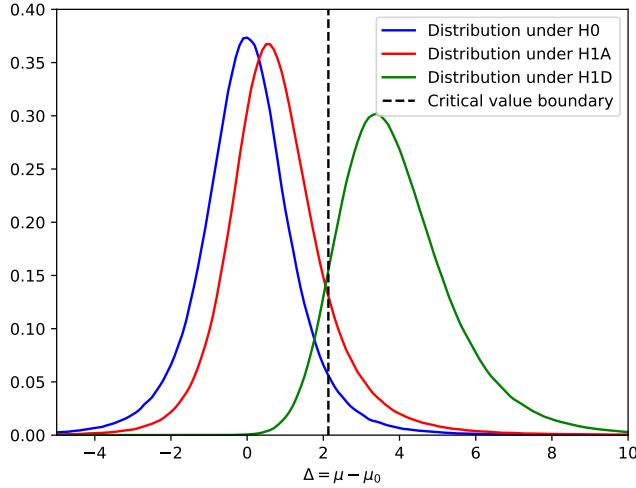


Figure 7.8: Numerically estimated distributions of the test statistic for various scenarios of values of the underlying parameter μ .

line 5 (note here that μ_0 represent the value under the null hypothesis, while μ represents the value of the actual underlying mean). Then, in lines 6 to 8, the corresponding test statistic is calculated via equation (7.10). This `tStat` function is used later to generate many separate test statistics.

- In line 11, the value of μ under the null hypothesis μ_0 is specified, along with the different values of μ for our two different scenarios, μ_{1A} , and μ_{1B} .
- In line 12 the standard deviation `sig`, and the number of sample observations for each sample group `n`, are specified.
- In line 13 the total number of sample groups (i.e. test statistics) that will be generated for each scenario is specified as `N`. In line 14, the significance level `alpha` is defined.
- In lines 16 to 20, the `tStat` function is used along with a series of comprehensions to generate `N` test statistics based on several different scenarios. In the first scenario `dataH0`, the actual underlying parameter matches that under the null hypothesis μ_0 . In the second scenario `dataH1A`, the underlying parameter has the value of μ_{1A} . In the third scenario `dataH1B`, the underlying parameter has the value of μ_{1B} . Scenarios `dataH1C` and `dataH1D` are similar to `dataH1B`, however for `dataH1C` the number of sample observations in the sample group is doubled, and for `dataH1D`, the number of sample observations in each group doubled, and the underlying standard deviation halved.
- In line 22, the critical value for the significance level `alpha` is calculated by using the `quantile()` function on a T-distribution `TDist()`, with $n-1$ degrees of freedom.
- In line 23 the function `estPwr` is defined, which takes an array of test statistics as input, and then approximates the corresponding power of the scenario as the proportion of statistics that exceed `tCrit` calculated previously (i.e. the proportion of cases for which the null hypothesis was rejected). Note the use of the `.>` which returns an array of `true`, `false` values, which are then summed up and divided by `N`.

- In lines 25 to 30, the value corresponding to the boundary of the rejection region is printed as output, along with the numerically estimated power of the hypothesis test under various actual values of μ . Note that as the actual parameter deviates further from that under the null hypothesis, the power of the test increases. Note also that as the number of sample observations in each group increases, and the standard deviation of the test decreases, the power of the test increases.
- In lines 32 to 34 the `kde()` function from the `KernelDensity` package is used to create three KDE type objects based on the arrays of the test statistics of lines 16, 17, and 20, for the scenarios of $\mu = \mu_0$, $\mu = \mu_{1A}$, and $\mu = \mu_{1B}$ respectively (note the last scenario also has standard deviation $\sigma/2$, and $2n$ sample observations in each sample group).
- In line 35 the range of means over which the KDE PDF's will be plotted is specified as `xGrid`.
- In lines 37 to 40, the KDE PDF's from lines 32 to 34 are plotted, along with the critical value of the rejection boundary. Note that the area under the PDF to the right of the critical value represent the power of the test for that scenario. Hence, the further the underlying parameter varies from the null hypothesis, the greater the power of the test.
- As a side point, note that the curves shown in Figure 7.8 could have alternatively been obtained via the *non-central T-distribution*.

From Listing 7.17, we can see that the statistical power of a hypothesis test can vary greatly, and depends not only on the parameters of the test, such as the number of observations in the sample group n and the specified sensitivity level α , but also on the underlying parameter values μ and σ . Hence, a key aspect of *experimental design* involves determining the test parameters such that not only is the probability of a type I error controlled, but that the test is sufficiently powerful over a range of different scenarios. This is important, as in reality there are an infinite number of H_1 's, any one of which could describe the underlying parameters. By designing a statistical test such that it has sufficient power, then we have confidence that if the underlying parameter deviates from the null hypothesis, then this will be identified.

A final example is presented in Listing 7.18 below where the concept of the *power curve* is introduced. In this example, a function which estimates the power of a one sided T-test, according to (7.44), is created. This function is then run for the same hypothesis test setup over a range of different values of μ , for various scenarios of different numbers of observations, of $n = 5, 10, 20, 30$. For each scenario, the power is estimated and the resulting power curves plotted. From the resulting Figure 7.9, it can be seen that as the number of sample observations increases, the statistical power of the test under each scenario increases.

Listing 7.18: Power curves for different sample sizes

```

1  using Distributions, KernelDensity, PyPlot
2
3  function tStat(mu0,mu,sig,n)
4      sample = rand(Normal(mu,sig),n)
5      xBar    = mean(sample)
6      s       = std(sample)
7      (xBar-mu0) / (s/sqrt(n))
8  end
9
10 function powerEstimate(mu0,mu1,sig,n,alpha,N)

```

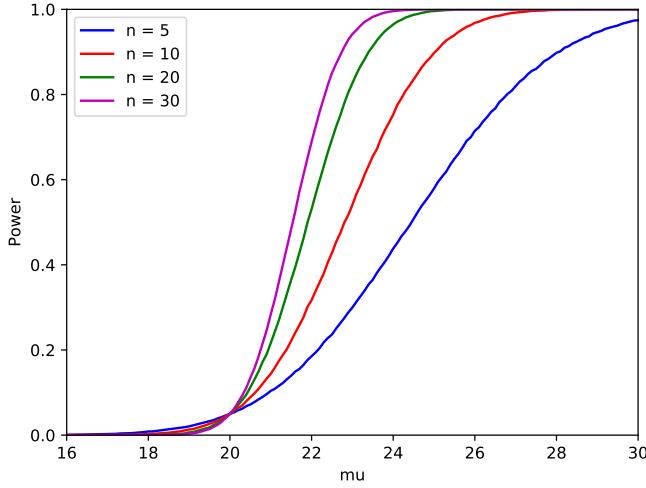


Figure 7.9: Power curves for the one-sided T-test with different sample sizes.

```

11      sampleH1 = [tStat(mu0,mu1,sig,n) for _ in 1:N]
12      critVal = quantile(TDist(n-1),1-alpha)
13      sum(sampleH1 .> critVal)/N
14  end
15
16  mu0 = 20
17  sig = 5
18  alpha = 0.05
19  N = 10^5
20  rangeMu1 = 16:0.1:30
21
22  powersN05 = [powerEstimate(mu0,mu1,sig,5,alpha,N) for mu1 in rangeMu1]
23  powersN10 = [powerEstimate(mu0,mu1,sig,10,alpha,N) for mu1 in rangeMu1]
24  powersN20 = [powerEstimate(mu0,mu1,sig,20,alpha,N) for mu1 in rangeMu1]
25  powersN30 = [powerEstimate(mu0,mu1,sig,30,alpha,N) for mu1 in rangeMu1]
26
27  plot(rangeMu1,powersN05, "b", label="n = 5")
28  plot(rangeMu1,powersN10, "r", label="n = 10")
29  plot(rangeMu1,powersN20, "g", label="n = 20")
30  plot(rangeMu1,powersN30, "m", label="n = 30")
31  xlim(minimum(rangeMu1) ,maximum(rangeMu1))
32  ylim(0,1)
33  legend()
34  xlabel("mu")
35  ylabel("Power")

```

- In lines 3 to 8, the function `tStat` is defined, which returns the value of the test statistic for a randomly generated group of sample observations. This function is identical to that created in Listing 7.17.
- In lines 10 to 14, the function `powerEstimate` is created, which uses a Monte Carlo approach to approximate the power of the one sided hypothesis test (7.44), given the value under the null hypothesis `mu0`, and the actual parameter of the underlying process `mu1`. The other arguments of the function include the number of sample observations in each group `n`, the

actual standard deviation `sig`, the chosen significance level `alpha`, and the total number of groups (i.e. test statistics) used in the Monte Carlo approximation `N`.

- In line 11 the function `tStat` is used along with a comprehension to generate `N` test statistics from `N` independent sample groups. The test statistics are then stored as the array `sampleH1`.
- In line 12, the analytic critical value for the given scenario of inputs is calculated in the same manner as in line 22 of Listing 7.17.
- In line 13 the proportion of test statistics greater than the critical value is calculated using the same approach as that of line 23 of Listing 7.17.
- In lines 16 to 20, the parameters of the problem are specified. The value under the null hypothesis `mu0`, the underlying variance of the unknown process `sig`, and the number of sample groups (i.e. test statistics) to be used in the Monte Carlo approach `N`. The range over which the underlying mean of the process `mu1` will be calculated is also specified as `rangeMu1`.
- In lines 22 to 25, four separate scenarios are considered, and for each, the power of the test over a range of different actual values of `mu1` are calculated. Note the only variable changed between each scenario is the number of sample observations in each group, `n`, in the Monte Carlo estimate.
- In lines 22 to 25 comprehensions are used along with the previously defined `pwrEstimate()` function to calculate the statistical power for four different scenarios. For each scenario, the power is calculated over the range of values of `mu1` in `rangeMu1`. Note that the only variable changed between each scenario is the number of sample observations `n`.
- In line 22 the `pwrEstimate` function is used to estimate the power of the hypothesis test, given 5 sample observations in each sample group, over the range of values of `mu1` in `rangeMu1`. The values of the approximated power are stored in the array `powersN05`.
- In lines 23, 24 and 25, the same approach as that of line 22 is used, however the number of sample observations, `n`, is increased to 10, 20, and 30 respectively. In each case, the power values are stored in the arrays shown.
- In lines 27 to 35, the resulting power curves are plotted in Figure 7.9. It can be seen that for `mu` values greater than `mu0=20`, as the number of observations in each sample group (`n`) increases, the power of the test (i.e. the probability of correctly rejecting H_0) increases. It can be seen that at $\mu_0=20$, the power of the tests correspond to 0.05, regardless of the number of sample observations in each sample group. This is expected, given the large number of sample groups used for the Monte Carlo estimate `N`.
- Another interesting point to note is that where $\mu < 20$, the ordering of the curves is reversed. For example, one can see that in this region the scenario where $n=30$ has less power than that for $n=5$, due to the fact that the probability of rejecting the null hypothesis at all, is less.
- Another point to note is that the x-axis could be scaled to represent the difference between the value of `mu0` under the null hypothesis, and the various possible values of `mu1`. Furthermore one could make the axis scale invariant by dividing said difference by the standard deviation. Such curves are often seen in experimental design reference material.

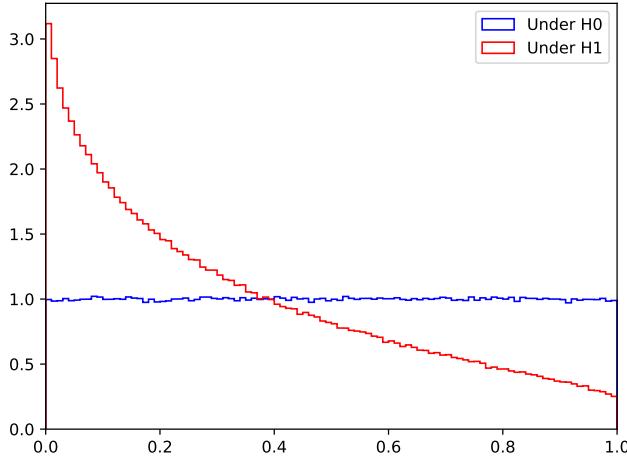


Figure 7.10: The distribution of the p -value under H_0 vs. the distribution of the p -value under a point in H_1 ($\mu = 22$).

Distribution of the p -value

In closing this chapter, we now discuss the concept of the distribution of the p -value. Throughout this chapter, equations of the form $p = \mathbb{P}(S > u)$ were presented, where S is a random variable representing the test statistic, u is the observed test statistic, and p is the p -value of the observed test statistic.

An alternative representation is to consider $P = 1 - F(S)$, where $F(\cdot)$ is the CDF of the test statistic under H_0 . Note that in this case, P is actually a random variable - a transformation of the test statistic random variable S . Assume that S is continuous, hence $\mathbb{P}(S < u) = F(u)$. We now have,

$$\begin{aligned}\mathbb{P}(P > x) &= \mathbb{P}(1 - F(S) > x) \\ &= \mathbb{P}(F(S) < 1 - x) \\ &= \mathbb{P}(S < F^{-1}(1 - x)) \\ &= F(F^{-1}(1 - x)) \\ t &= 1 - x.\end{aligned}$$

Recalling that for a uniform(0,1) random variable, the CCDF is $1 - x$ on $x \in [0, 1]$. Therefore under H_0 , P is a uniform(0,1) random variable.

This fact is demonstrated in Listing 7.19, where we consider a situation of a T-test under two different scenarios, firstly under $H_0 : \mu = 20$ and then under a point in H_1 , where $\mu = 22$. Through a Monte Carlo approach, N p -values are generated for the two scenarios, and their resulting numerically estimated distributions are then plotted in Figure 7.10. The results illustrate that under H_0 the distribution of p is uniform, yet under H_1 it is not.

Listing 7.19: Distribution of the p -value

```

1  using Random, Distributions, KernelDensity, PyPlot
2  Random.seed!(1)
3
4  function pval(mu0,mu,sig,n)
5      sample = rand(Normal(mu,sig),n)
6      xBar    = mean(sample)
7      s       = std(sample)
8      tStat   = (xBar-mu0) / (s/sqrt(n))
9      ccdf(TDist(n-1), tStat)
10 end
11
12 mu0, mu1  = 20, 22
13 sig, n, N = 7, 5, 10^6
14
15 pValsH0 = [pval(mu0,mu0,sig,n) for _ in 1:N]
16 pValsH1 = [pval(mu0,mu1,sig,n) for _ in 1:N]
17
18 plt[:hist](pValsH0,100, normed="true", histtype="step", color="b", label="Under H0")
19 plt[:hist](pValsH1,100, normed="true", histtype="step", color="r", label="Under H1")
20 xlim(0,1)
21 legend(loc="upper right")

```

- In lines 4 to 10 the function `pval()` is defined. This function is similar to the `tStat` function from Listings 7.17 and 7.18, but includes the extra line 9, which calculates the *p*-value from test statistic of line 8. Note the use of the `ccdf()` function.
- In lines 12 and 13 the parameters of the problem are defined. The value of μ under both the null hypothesis and a point in H_1 are defined as `mu0` and `mu1` respectively, while the standard deviation, number of sample observations per group, and number of *p*-values to be generated are defined as `sig`, `n` and `N` respectively.
- In lines 15 and 16 the `pval()` function is used along with comprehensions to calculate N *p*-values under the two scenarios of `mu0` and `mu1`, with the results stored in the arrays `pValsH0` and `pValsH1`.
- In lines 18 and 19 histograms of the *p*-values stored in `pValsH0` and `pValsH1` are plotted. The results demonstrate that the distribution of the *p*-value is dependent on the underlying value of μ .

Chapter 8

Linear Regression - DRAFT

We now explore one of the most popular statistical techniques in practice, *regression analysis*. The key idea of regression analysis and supervised learning in general is to consider a so-called *dependent variable* Y and see how it is affected by one or more *independent variables*, typically denoted X . That is, regression analysis considers how X affects Y . In contrast, unsupervised learning only involves X . Such cases were handled in the context of confidence intervals and hypothesis tests in the previous two chapters, mostly for a single variable X .

When considering Y and X as random variables (with X possibly vector valued), the term *regression* of Y on X signifies the *conditional expectation* of Y , given an observed value of X , say $X = x$. That is, one may stipulate that both X and Y are random, and, given some observed value x of X , then the regression is given by,

$$\hat{y} = \mathbb{E}[Y | X = x]. \quad (8.1)$$

Here, \hat{y} is a *predictor* of the dependent variable Y , given an observation of the independent variable X . The simplest and most widely studied regression example assumes that the regression function is affine (i.e. linear) in nature. That is,

$$\hat{y} = \mathbb{E}[Y | X = x] = \alpha + \beta x,$$

where α and β describe the intercept and slope respectively of a line. In this case, a typical model is, $Y = \alpha + \beta x + \epsilon$, where ϵ is considered a noise term, typically taken as a normally distributed random variable independent of everything else, with a variance that does not depend on x .

A widely used method of finding α and β is via *least squares*. Given a series of observation tuples, $(x_1, y_1), \dots, (x_n, y_n)$, which can be viewed as a “cloud of points”, the least squares method finds the so-called “line of best fit”, $\hat{y} = \hat{\alpha} + \hat{\beta}x$, where $\hat{\alpha}$ and $\hat{\beta}$ are estimates of α and β obtained via least squares from the data.

The concept of least squares, along with many associated regression concepts is covered in detail in this chapter. Furthermore, several extensions are also covered, many of which are used commonly in practice. From a software perspective, the key tool used in this chapter is the Julia *GLM package*.

This chapter is structured as follows: In Section 8.1 we focus on least squares. In Section 8.2 we present the basic linear regression model with one variable. In Section 8.3 we move onto multiple

linear regression. In Section 8.4 we explore further model adaptations such as non-linear transformations and working with categorical variables. We close with Section 8.5 dealing with classical methods of model selection as well as a brief exploration of the LASSO method (also further discussed in the next chapter).

8.1 Clouds of Points and Least Squares

To begin, consider a sequence of observations, $(x_1, y_1), \dots, (x_n, y_n)$, which, when plotted on the Cartesian plane, yields a *cloud of points*. Then, assuming that a functional relationship exists between x and y , such as $y = f(x)$, the first goal is to use these points to estimate the function $f(\cdot)$.

A classic non-statistical way of obtaining $f(\cdot)$ assumes that the observations exactly follow $y_i = f(x_i)$ for every i . This requires assuming that there are no two y values which share the same x value. A common assumption is that $f(\cdot)$ is a polynomial of order $n - 1$, and based on this assumption *polynomial interpolation* can be carried out. This involves seeking the coefficients, c_0, \dots, c_{n-1} of the polynomial,

$$f(x) = c_{n-1}x^{n-1} + \dots + c_2x^2 + c_1x + c_0.$$

The coefficients can be found by constructing a *Vandermonde matrix* as shown in (8.2) below, and then solving the coefficients (c_0, \dots, c_{n-1}) of the linear system,

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ 1 & x_3 & x_3^2 & \cdots & x_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix}. \quad (8.2)$$

Consider now, for example, the 6 data points shown in Figure 8.1. In this figure, a fifth degree polynomial is fit to these points, shown in blue. However, although the polynomial perfectly fits the data points, one can argue that a linear approximation may be a better fit of the data.

From Figure 8.1, one can see that implementing a polynomial interpolation approach for data fitting can be highly problematic, since, by requiring every point to agree with $f(\cdot)$ exactly, the approach often results in an “over-fit” model. This phenomenon, known as *over fitting*, is common throughout data-analysis, since it is often possible to find a model that describes the observed data exactly, but when new observations are made performs poorly. In addition, such models are often over-complicated for the scenario at hand.

In Listing 8.1 below, polynomial interpolation is carried out by constructing a Vandermonde matrix, which is then used to solve for the coefficients c_i . The resulting polynomial fit to the data (with coefficients rounded) is,

$$y = -0.01x^5 + 0.26x^4 - 2.73x^3 + 8.96x^2 + 6.2x - 42.72.$$

This polynomial is then plotted against the line,

$$y = 4.58 + 0.17x$$

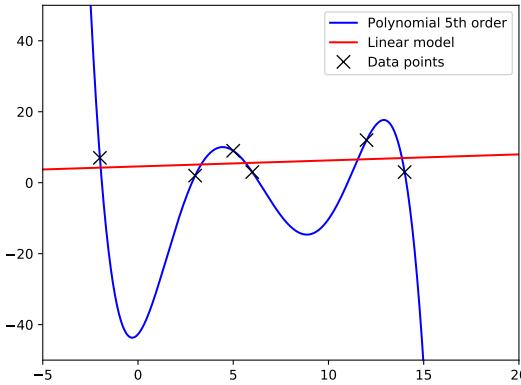


Figure 8.1: Cloud of point fitting via a fifth degree polynomial, and a first degree polynomial (linear model). Although the higher order polynomial fits the data perfectly, one could argue it is not better in practice.

and displayed in Figure 8.1. The parameters of this line were obtained via the least squares method, which is described in the next subsection. One can see that although the polynomial fits the data exactly, it is much more complicated than the line and appears to be overfitting the data. For example, if a seventh observation was recorded, then the line may be a far better predictor.

Listing 8.1: Polynomial interpolation vs. a line

```

1  using PyPlot
2
3  xVals = [-2, 3, 5, 6, 12, 14]
4  yVals = [7, 2, 9, 3, 12, 3]
5  n = length(xVals)
6
7  V = [xVals[i+1]^(j) for i in 0:n-1, j in 0:n-1]
8  c = V\yVals
9  xGrid = -5:0.01:20
10 f1(x) = c'*[x^i for i in 0:n-1]
11
12 beta0, betal = 4.58, 0.17
13 f2(x) = beta0 + betal*x
14
15 plot(xGrid,f1.(xGrid),"b",label="Polynomial 5th order")
16 plot(xGrid,f2.(xGrid),"r",label="Linear model")
17 plot(xVals,yVals,"kx",ms="10",label="Data points")
18 xlim(-5,20)
19 ylim(-50,50)
20 legend(loc="upper right")

```

- In line 7 the matrix V is defined, which represents the Vandermonde matrix as shown in (8.2).
- In line 8 the \backslash operator is used to solve the system of equations shown in (8.2), returning the coefficients as an array, which is then stored as c .
- Line 10 the function $f1()$ is defined, which uses the inner product by multiplying c' with an array of monomials, and describes our polynomial of order $n-1$.

- Line 13 the function `f2()` is defined, which describes our linear model. Note the use of hard coded coefficients here.
- Notice the use of mapping `f1()` and `f2()` over `xGrid` via the broadcast operator “.” in lines 15 and 16.

Fitting a Line Through a Cloud of Points

Although a line of the form $y = \beta_0 + \beta_1 x$ may be a sensible model for a series of cloud points, $y = f(x)$, it is obvious that $y_i = f(x_i)$ will not be satisfied for many, or all of the observations. The question then arises how to best select β_0 and β_1 ?

The typical approach is to select β_0 and β_1 such that the deviations between y_i and \hat{y}_i are minimized, where,

$$\hat{y}_i = \beta_0 + \beta_1 x_i.$$

Importantly, there is no universal way for measuring such deviations, instead there are several different measures. Here the two most common measures are presented; the *L₂ norm* (or *Euclidean norm*) based measure, and the *L₁ norm* based measure, both of which are defined below,

$$L^{(1)} := \sum_{i=1}^n |\hat{y}_i - y_i|, \quad L^{(2)} := \sum_{i=1}^n (\hat{y}_i - y_i)^2.$$

Both of these values are based on the elements, e_1, \dots, e_n where $e_i := \hat{y}_i - y_i$ (also known as the *errors*, or *residuals*). The first is the *L₁* norm of these values and the second is the *L₂* norm (i.e the square) of these values.

Observe that, if the data is considered fixed, both $L^{(1)}$ and $L^{(2)}$ depend on β_0 and β_1 . Hence estimates of these coefficients can be obtained via,

$$\min_{\beta_0, \beta_1} L^{(\ell)}, \tag{8.3}$$

where ℓ is either 1 or 2. In practice, the most common and simplest method is to focus on $\ell = 2$. For many reasons this is due to analytical tractability of the *L₂* norm, and minimization of the *L₂* norm is presented via least squares later in this section. However, at this point, both $\ell = 1$ and $\ell = 2$ are considered, and the optimization for equation (8.3) carried out via naive Monte Carlo guessing. This is done in order to understand the differences between $\ell = 1$ and $\ell = 2$ qualitatively. The remainder of the chapter then focuses solely on $\ell = 2$, with $\ell = 1$, and other loss measures, left to further reading.

In Listing 8.2 below, uniform random values over the grid $[0, 5] \times [0, 5]$ are trialled for β_0 and β_1 . For each pair of values, the *L₁* and *L₂* costs are compared to their previous values, and, if the costs are lower, then the corresponding values for β_0 and β_1 adopted. By repeating this process N times, then for large N , we aim to obtain coefficient values that closely approximate those that minimize *L₁* and *L₂*. Note, for clarity in this example, β_0 and β_1 are denoted as `alpha` and `beta` respectively. The results are presented in Figure 8.2. Pictorially, the summands of the costs $L^{(1)}$ and $L^{(2)}$ are also presented. In the $L^{(1)}$ case these are presented as lines, while in the $L^{(2)}$ case these are presented as squares, hence the name “least squares”.

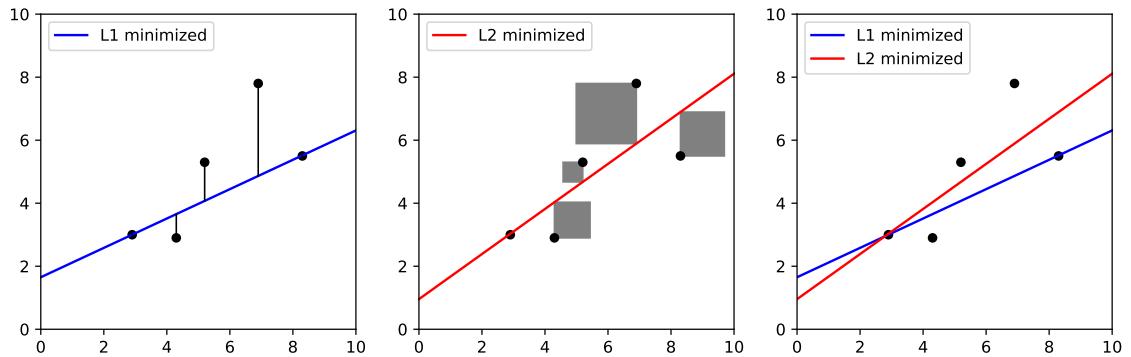


Figure 8.2: The blue line minimizes L_1 (i.e the sum of black lines), while the red line minimizes L_2 (i.e the sum of squares).

Listing 8.2: L1 and L2 norm minimization by MC Simulation

```

1  using DataFrames, Distributions, PyPlot, PyCall, Random, LinearAlgebra, CSV
2
3  @pyimport matplotlib.patches as patch;
4  @pyimport matplotlib.lines as line
5  Random.seed!(0)
6  data = CSV.read("L1L2data.csv")
7  xVals, yVals = data.X, data.Y
8  n, N = 5 , 10^6
9  alphaMin, alphaMax, betaMin, betaMax = 0, 5, 0, 5
10 alpha1, beta1, alpha2, beta2, bestL1Cost, bestL2Cost = 0.0, 0.0, 0.0, 0.0, Inf, Inf
11 for _ in 1:N
12     rAlpha, rBeta=rand(Uniform(alphaMin,alphaMax)),rand(Uniform(betaMin,betaMax))
13     L1Cost = norm(rAlpha .+ rBeta*xVals - yVals,1)
14     if L1Cost < bestL1Cost
15         alpha1 = rAlpha
16         beta1 = rBeta
17         bestL1Cost = L1Cost
18     end
19     L2Cost = norm(rAlpha .+ rBeta*xVals - yVals)
20     if L2Cost < bestL2Cost
21         alpha2 = rAlpha
22         beta2 = rBeta
23         bestL2Cost = L2Cost
24     end
25 end
26
27 fig = figure(figsize=(12,4))
28 ax1 = fig[:add_subplot](1,3,1)
29 ax1[:set_aspect]("equal")
30 plot(xVals,yVals,"k.",ms=10)
31 plot([0,10],[alpha1, alpha1 .+ beta1*10],"b",label="L1 minimized")
32 legend(loc="upper left")
33 xlim(0,10); ylim(0,10)
34
35 ax2 = fig[:add_subplot](1,3,2)
36 ax2[:set_aspect]("equal")

```

```

37 plot(xVals,yVals,"k.",ms=10)
38 plot([0,10],[alpha2, alpha2 .+ beta2*10],"r",label="L2 minimized")
39 legend(loc="upper left")
40 xlim(0,10); ylim(0,10)
41
42 ax3 = fig[:add_subplot](1,3,3)
43 ax3[:set_aspect]("equal")
44 plot(xVals,yVals,"k.",ms=10)
45 plot([0,10],[alpha1, alpha1 .+ beta1*10],"b",label="L1 minimized")
46 plot([0,10],[alpha2, alpha2 .+ beta2*10],"r",label="L2 minimized")
47 legend(loc="upper left")
48 xlim(0,10); ylim(0,10)
49
50 d = yVals - (alpha2 .+ beta2*xVals)
51 for i in 1:n
52     x,y = xVals[i],yVals[i]
53     l=line.Line2D([x, x], [y, alpha1 .+ beta1*x], lw=1,color="black")
54     r=patch.Rectangle([x,y],-d[i],-d[i],lw=1,ec="black",fc="black",alpha=0.5)
55     ax1[:add_artist](l);ax2[:add_artist](r)
56 end
57 println("L1 line: $(round(alpha1,digits = 2)) + $(round(beta1,digits = 2))x")
58 println("L2 line: $(round(alpha2,digits = 2)) + $(round(beta2,digits = 2))x")

```

L1 line: 1.65 + 0.47x
L2 line: 0.96 + 0.72x

- In line 5 the observation data is stored in `xVals` and `yVals`.
- Lines 11–23 search over random alpha and beta values in the interval $[0, 5]$. Each time a random set of values is generated, it is checked to see if it improves $L^{(1)}$ and $L^{(2)}$ and if it is better than the value we had before, that new value is stored, along with the corresponding values of `beta0` and `beta1`. Note the use of `norm()` with 1 indicating an L_1 norm and no parameter indicating the default L_2 norm.
- Lines 25–43 plot the curves and data points in a rather straightforward manner. The figures are kept with handles `ax1` and `ax2` for the code that follows.
- Lines 45–51 plot the residuals as line segments for the $L^{(1)}$ case (code line 48) and as squares for the $L^{(2)}$ case (code line 49). The actual graphics primitives, `l` and `r`, are added to the figures in line 50.

Least Squares

Having explored the fact that there are multiple ways to fit a line through a cloud of points, we now focus on the most common and mathematically simple way, the method of *least squares*. This method involves finding the values of β_0 and β_1 that minimize L_2 . Note that that the loss function L_2 can be written in several alternative ways, such as,

$$L^{(2)} = \sum_{i=1}^n e_i^2 = \sum_{i=1}^n (y_i - \hat{y}_i)^2 = \sum_{i=1}^n (y_i - \beta_0 - \beta_1 x_i)^2 = \|y - A\beta\|^2. \quad (8.4)$$

Note that the last representation is the most general, where β represents a vector of all coefficients $\beta = (\beta_0, \beta_1, \dots, \beta_{p-1})$, y a vector of all observations $y = (y_1, y_2, \dots, y_n)$, and A the *design matrix* $A \in \mathbb{R}^{n \times p}$. Further, for a vector, z , we denote by $\|z\|^2$ the value, $\sum_{i=1}^n z_i^2$. Following along with our introductory case where there is only one independent variable, i.e. $p = 2$, then $\beta = (\beta_0, \beta_1)$, and the design matrix is given by,

$$A = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}. \quad (8.5)$$

In general however, A may be any matrix, with $p \leq n$, where p is the number of coefficients in the model, and n the number of observations. With such a matrix A at hand, and with observation outcomes y present, the problem of finding β so as to minimize $\|y - A\beta\|^2$ is called the *least squares problem*.

It turns out that the theory of least squares is simplest when A is a *full rank* matrix. That is, when its rank equals p , i.e. all the columns are linearly independent. In this case, the *Graham matrix*, $A'A$ is *non-singular*, and the *Moore-Penrose pseudo-inverse* exists, and is defined as follows,

$$A^\dagger := (A'A)^{-1}A'. \quad (8.6)$$

Then, it follows from the theory of linear algebra that,

$$\hat{\beta} = A^\dagger y, \quad (8.7)$$

minimizes $L^{(2)}$. An alternative representation can be shown by considering the *QR factorization* of A , and denoting $A = QR$, where Q is a matrix of orthonormal columns and R is an upper triangular matrix. In this case, it is easy to see that,

$$A^\dagger = R^{-1}Q'. \quad (8.8)$$

Further, even if A is not full rank, we may compute the pseudo-inverse by considering the *singular value decomposition* of A . Here $A = U\Sigma V'$ where U is a $n \times n$ matrix orthogonal, Σ is an $n \times 2$ matrix with non-zero elements only on the diagonal, and V is a 2×2 orthogonal matrix. In such a case,

$$A^\dagger = V\Sigma^+U'. \quad (8.9)$$

In general, for solving least squares, it is easiest to make use of the powerful Julia *backslash* operator, \backslash . This notation, popularized by precursor languages such as Matlab, treats $Ab = y$ as a general system of equations and allows to write $b = A \backslash y$ as the “solution” for b . If A happens to be square and non-singular then analytically, it is equivalent to coding $b = \text{inv}(A) * y$. However, from a numerical and performance perspective, use of backslash is generally preferred as it calls upon dedicated routines from LAPAK. This is the linear algebra package, initially bundled into Matlab, but also employed by Julia and a variety of other scientific computing systems. More importantly for our case, when A is skinny ($p < n$) and full rank, evaluation of $b = A \backslash y$, produces the least squares solution. I.e. it sets, $b = A^\dagger y$ in a numerically efficient manner.

There are many ways to derive the optimal $\hat{\beta}$ of (8.7). Another straightforward approach is to consider,

$$L^{(2)} = \sum_{i=1}^n \left(\sum_{j=1}^p A_{ij} \beta_j - y_i \right)^2.$$

Here, for convenience, consider the indexes of the vector β as running from 1 to p (instead of 0 to $p-1$). By treating $L^{(2)}$ as a function of β_1, \dots, β_p we can evaluate its gradient, by calculating the derivative with respect to each β_k as follows,

$$\begin{aligned} \frac{\partial L^{(2)}}{\partial \beta_k} &= 2 \sum_{i=1}^n A_{ik} \left(\sum_{j=1}^n A_{ij} x_j - y_i \right) \\ &= 2 \sum_{i=1}^n (A')_{ki} (Ax - y)_i \\ &= \left(2A'(A\beta - y) \right)_k. \end{aligned}$$

Hence the gradient is $\nabla L^{(2)} = 2A'(A\beta - y)$. Equating this to zero, the so-called *normal equations* can be obtained,

$$A'A\beta = A'y, \quad (8.10)$$

and for the specific A of (8.5), the normal equations read as,

$$\begin{aligned} n\hat{\beta}_0 + \hat{\beta}_1 \sum_{i=1}^n x_i &= \sum_{i=1}^n y_i, \\ \hat{\beta}_0 \sum_{i=1}^n x_i + \hat{\beta}_1 \sum_{i=1}^n x_i^2 &= \sum_{i=1}^n y_i x_i. \end{aligned}$$

These are called the *least squares normal equations*, and the solution to these results in the *least squares estimators* $\hat{\beta}_0$ and $\hat{\beta}_1$. Using the sample means, \bar{x} and \bar{y} the estimators are,

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}, \quad \hat{\beta}_1 = \frac{n \sum_{i=1}^n y_i x_i - \left(\sum_{i=1}^n y_i \right) \left(\sum_{i=1}^n x_i \right)}{n \sum_{i=1}^n x_i^2 - \left(\sum_{i=1}^n x_i \right)^2}. \quad (8.11)$$

In a different format, the following quantities are also commonly used,

$$S_{xx} = \sum_{i=1}^n (x_i - \bar{x})^2, \quad S_{xy} = \sum_{i=1}^n (y_i - \bar{y})(x_i - \bar{x}).$$

These yield an alternative formula for $\hat{\beta}_1$,

$$\hat{\beta}_1 = \frac{S_{xy}}{S_{xx}}. \quad (8.12)$$

Finally, one may also use the sample correlation, and sample standard deviations to obtain $\hat{\beta}_1$,

$$\hat{\beta}_1 = \text{corr}(x,y) \frac{\text{std}(y)}{\text{std}(x)}. \quad (8.13)$$

In Listing 8.3, each of the representations covered above is now used to obtain the least squares estimate for the same dataset used in Listing 8.2. The purpose is to illustrate that a variety of alternative methods, representations and commands can be used to solve least squares. The comments that follow the listing help add more insight to each method, and in addition shed light on an additional gradient descent method employed. In total 10 different methods are used, labelled A to J, and each one obtains the same estimates for β_0 and β_1 from the data. Approaches A and B use the formulas above for the case of $p = 2$ (simple linear regression). Approaches C, D, E, F, G and H work with either the normal equations, (8.10), or the pseudo inverse, A^\dagger . Approach I executes a gradient descent algorithm (see code comments below). Finally, J and K call upon the GLM statistical package, which is covered in more detail in Section 8.2. From the output, it can be seen that all approaches yield the same estimates.

Listing 8.3: Computing least squares estimates

```

1  using DataFrames, GLM, Statistics, LinearAlgebra, CSV
2
3  data = CSV.read("L1L2data.csv")
4  xVals, yVals = Array{Float64}(data.X), Array{Float64}(data.Y)
5  n = length(xVals)
6  A = [ones(n) xVals]
7
8  # Approach A
9  xBar,yBar = mean(xVals),mean(yVals)
10 sXX, sXY = ones(n)'*(xVals.-xBar).^2, dot(xVals.-xBar,yVals.-yBar)
11 b1A = sXY/sXX
12 b0A = yBar - b1A*xBar
13
14 # Approach B
15 b1B = cor(xVals,yVals)*(std(yVals)/std(xVals))
16 b0B = yBar - b1B*xBar
17
18 # Approach C
19 b0C,b1C = A'A \ A'yVals
20
21 # Approach D
22 Adag = inv(A'*A)*A'
23 b0D,b1D = Adag*yVals
24
25 # Approach E
26 b0E,b1E = pinv(A)*yVals
27
28 # Approach F
29 b0F,b1F = A\yVals
30
31 # Approach G
32 F = qr(A)
33 Q, R = F.Q, F.R
34 b0G,b1G = (inv(R)*Q')*yVals
35
36 # Approach H
37 F = svd(A)
38 V, Sp, Us = F.V, Diagonal(1 ./ F.S), F.U'
39 b0H,b1H = (V*Sp*Us)*yVals
40
41 # Approach I
42 eta,eps = 0.002,10^-6.
43 b,bPrev = [0,0], [1,1]
```

```

44 while norm(bPrev-b) > eps
45     bPrev = b
46     b = b - eta*2*A'*(A*b - yVals)
47 end
48 b0I,b1I = b[1],b[2]
49
50 # Approach J
51 modelJ = lm(@formula(Y ~ X), data)
52 b0J,b1J = coef(modelJ)
53
54 # Approach K
55 modelK = glm(@formula(Y ~ X), data, Normal())
56 b0K,b1K = coef(modelK)
57
58 println(round.([b0A,b0B,b0C,b0D,b0E,b0F,b0G,b0H,b0I,b0J,b0K], digits=3))
59 println(round.([b1A,b1B,b1C,b1D,b1E,b1F,b1G,b1H,b1I,b1J,b1K], digits=3))

```

```
[0.945, 0.945, 0.945, 0.945, 0.945, 0.945, 0.945, 0.945, 0.944, 0.945, 0.945]
[0.716, 0.716, 0.716, 0.716, 0.716, 0.716, 0.716, 0.716, 0.717, 0.716, 0.716]
```

- Observe that in line 5, we construct the design matrix, A .
- Lines 8–11 implement (8.11) using (8.12). For variety look at line 9. There we use an inner product with `ones(n)` for the first element, s_{XX} . Then for the second element we use the `dot()` function which takes the inner product of both its arguments.
- Lines 14–15 implement (8.13). This uses the built in `cor()` and `std()` functions.
- Line 18 is a direct solution of the normal equations (8.10). Here we use the backslash operator to solve the equations. The expression $A'A \setminus A'yVals$ is an array with the solution. But we transform it into the individual elements, $b0C$ and $b1C$.
- Lines 21 and 22, do the same thing, by finding A^\dagger , denoted `Adag` in line 21. Then it applied (as a linear transformation) to `yVals` in line 22.
- Line 25 shows the use of the `pinv()` function that computes A^\dagger directly.
- Line 28 computes $\hat{\beta}$ by using the built in backslash (`\`) operator. This delegates the exact numerical aspect to Julia, as opposed to forcing it directly as in the previous lines. It is generally the preferred method.
- Lines 31–33 use QR-factorization. In line 31 the object `F` is assigned the result of `qrfact`. Then the specific `Q` and `R` matrices are obtained from that object via `F[:Q]` and `F[:R]` respectively. Lines 33 then implements (8.8), representing A^\dagger via the code `inv(R)*Q'`.
- Lines 36–42 implement a completely different approach: *gradient descent*. Here the gradient, $\nabla L^{(2)}$ as described above is implemented via `2*A'* (A*b-yVals)` as in line 40. Gradient descent then iterates via,

$$b(t+1) = b(t) - \eta \nabla L^{(2)}.$$

The parameter η is known (in this context) as the *learning rate*. In the code we set it to 0.002 (line 36). The algorithm then iterates until the difference between two iterates, $b(t+1)$ and $b(t)$ is less than or equal to 10^{-6} . We explore a variant of gradient descent below.

- Lines 45-50 use the `lm()` and the `glm()` functions from the GLM package. The result is in a model object, denoted `modelI` and `modelJ` in the code. Then the `coef()` function (also from the GLM package) retrieves the estimates from the model objects. We elaborate much more on GLM in the sequel.

Stochastic Methods

When dealing with huge data-sets (not the primary focus of this book), one often tries to use alternative methods for solving least squares problems. In cases, where the points are of the form $(x_1, y_1), \dots (x_n, y_n)$, with both x_i and y_i scalar, this is typically not critical, even if n is in the order of millions. However, in more general situations (some of which described in Section 8.2) we have that each x_i is a high-dimensional p -vector. In such cases, carrying out least squares as described above is sometimes not numerically tractable.

For this, other methods, mostly popularized in machine-learning can sometimes be employed. The most basic of which is *stochastic gradient descent* (SGD). These methods have also been widely employed in other models, mostly popularly *deep neural networks*.

While SGD and related algorithms is far from the core focus of our book, we do present a simple SGD example, attempting to solve a least squares problem. In practice, you would not use SGD for such a simple problem.

Listing 8.4: Using SGD for least squares

```

1  using PyPlot, Random
2
3  n = 10^3
4  beta0 = 2.0
5  beta1 = 1.5
6  sigma = 2.5
7
8  Random.seed!(1958)
9  xVals = rand(0:0.01:5,n)
10 yVals = beta0 .+ beta1*xVals + rand(Normal(0,sigma),n)
11
12 pts = []
13 eta = 10^-3.
14 b = [0,0]
15 push!(pts,b)
16 for k in 1:10^4
17     i = rand(1:n)
18     g = [ 2(b[1] + b[2]*xVals[i]-yVals[i]),
19            2*xVals[i]*(b[1] + b[2]*xVals[i]-yVals[i])  ]
20     b = b - eta*g
21     push!(pts,b)
22 end
23
24 figure(figsize=(10,5))
25 subplot(121)
26 plot(first.(pts),last.(pts),"k",lw="0.5",label="SGD path")
27 plot(b[1],b[2],".b",ms="10.0",label="SGD")
28 plot(beta0,beta1,".r",ms="10.0",label="Actual")
29 xlim(0,2.5)

```

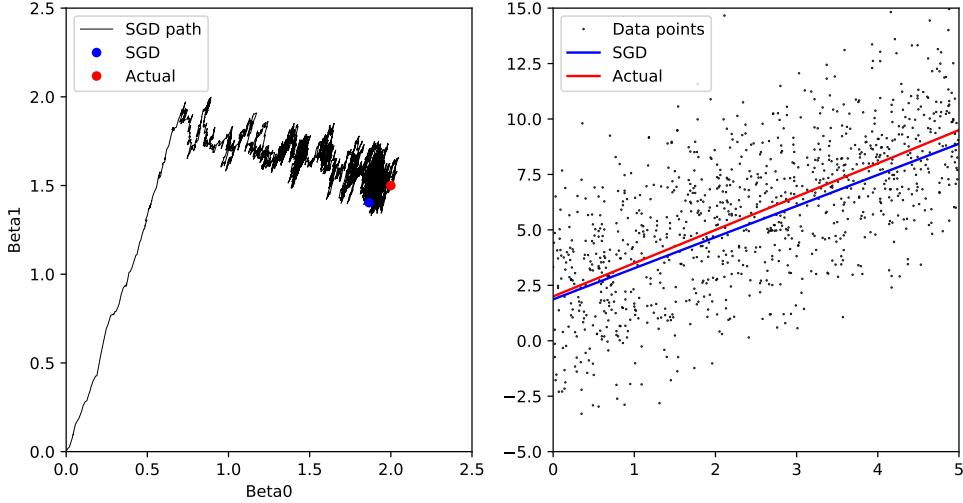


Figure 8.3: An application of stochastic gradient descent for solving least squares. Left: The path starting at $(0,0)$ ends at the blue point while the red point is the actual one. Right: The data with the fit line.

```

30 ylim(0,2.5)
31 legend(loc="upper left")
32 xlabel("Beta0")
33 ylabel("Beta1")
34
35 subplot(122)
36 plot(xVals,yVals,"k.",ms="1",label="Data points")
37 plot([0,5],[b[1],b[1]+5b[2]],"b",label="SGD")
38 plot([0,5],[beta0,beta0+5*beta1],"r",label="Actual")
39 xlim(0,5)
40 ylim(-5,15)
41 legend(loc="upper left");

```

- Lines 3–10 setup synthetic data for this problem. The x-values fall uniformly over the discrete grid $0 : 0.01 : 5$ and for every x-value, the y-value follows $y = \beta_0 + \beta_1 x + \epsilon$ where ϵ is normally distributed with a standard deviation of 2.5.
- Lines 12–22 implement stochastic gradient descent for 10^4 iterations. The starting value is $(\beta_0, \beta_1) = (0, 0)$. The learning rate is 10^{-3} . For plotting purposes, every additional point is pushed into the array, pts. The index i for the random data observation of each iteration is obtained in line 17. Then lines 18 and 19 evaluate the gradient with respect to that observation. Finally, line 20 makes the step in the direction g.
- Lines 24–28 plot the trajectory of the algorithm in parameter space. The correct value of β_0 and β_1 is plotted in red. The final value of the SGD is plotted in blue.
- Lines 30–33 plot the data points and the estimated line of best fit.

8.2 Linear Regression with One Variable

Having explored the notion of the line of best fit and least squares in the previous section, we now move onto the most basic statistical application: *linear regression with one variable* also known as *simple linear regression*. This is the case where we assume the following relationship between the random variables Y_i and X_i :

$$Y_i = \beta_0 + \beta_1 X_i + \epsilon_i. \quad (8.14)$$

Here ϵ_i is normally distributed with zero mean and variance σ^2 and is assumed independent across observation indexes i . By assuming such a specific form of the error term, ϵ , we are able to say more about the estimates of β_0 and β_1 than we did in the previous section. While least squares procedures gave us a solid way to obtain $\hat{\beta}_0$ and $\hat{\beta}_1$, by themselves, such procedures don't give any information about the reliability of the estimates. Hence by assuming a model such as above we can go further: We can carry out statistical inference for the unknown parameters, β_0, β_1 and σ^2 . This includes confidence intervals and hypothesis tests.

In this *linear regression* context, while we denote Y_i and X_i as random variables in (8.14), we assume that the Y values are random and the X values are observed and not random. We then obtain estimates of β_0, β_1 and σ^2 given the sample, $(x_1, y_1), \dots, (x_n, y_n)$. The x_i coordinates, assumed deterministic, are sometimes called the *design*. The y_i coordinates are assumed as observations following the relationship in (8.14) given $X = x_i$.

The standard statistical way to estimate β_0, β_1 and σ^2 is to use maximum likelihood estimation, see Section 5.4. In this case, using the normality assumption, the log-likelihood function is

$$\ell(\beta_0, \beta_1, \sigma^2 | y, x) = -\frac{n}{2} \log 2\pi - \frac{n}{2} \log \sigma^2 - \frac{1}{\sigma^2} \sum_{i=1}^n (y_i - \beta_0 - \beta_1 x_i)^2. \quad (8.15)$$

It can then be shown to be optimized by the same least squares estimates presented in the previous section. See for example formula, (8.11). Further, the optimizer for σ^2 is

$$\hat{\sigma}^2 = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i)^2. \quad (8.16)$$

With estimators such as $\hat{\beta}_0, \hat{\beta}_1$ and $\hat{\sigma}^2$ at hand, we can now carry out statistical inference for the regression model. The least squares estimators $\hat{\beta}_0$ and $\hat{\beta}_1$ are unbiased estimators for β_0 and β_1 respectively. However it turns out that $\hat{\sigma}^2$ is a slightly biased estimator of σ^2 . An unbiased estimator, denoted MSE (for Mean Square Error) is,

$$\text{MSE} = \frac{SS_{\text{residuals}}}{n-2}, \quad \text{with} \quad SS_{\text{residuals}} = \sum_{i=1}^n (y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i)^2. \quad (8.17)$$

The denominator, $n-2$ is called the *degrees of freedom*. It is also sensible to consider $SS_{\text{residuals}}$ in comparison to $SS_{\text{total}} = \sum_{i=1}^n (y_i - \bar{y})^2$. The former measures variation of residuals around the regression line and the latter measures total variation of the dependent variable. Sums of squares decompositions hold for linear regression as they do for ANOVA (see Section 7.3). This also motivates computing the quantity “*R squared*” defined as:

$$R^2 = 1 - \frac{SS_{\text{residuals}}}{SS_{\text{total}}} \in [0, 1].$$

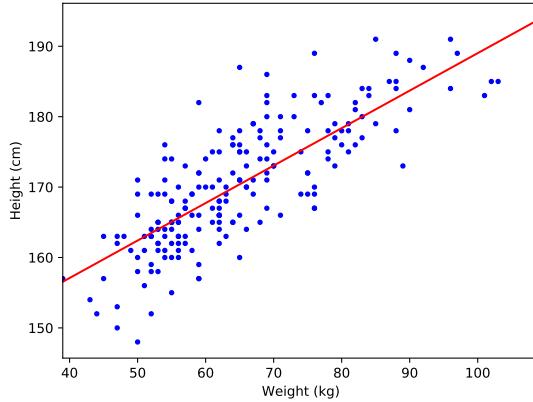


Figure 8.4: A scatter plot of Height vs. Weight with a line of best fit obtained via linear regression.

If R^2 is close to 1 then it implies that the residual variation is low whereas if R^2 is close to 0 it implies that most of the total variation is due to residuals. Considering R^2 as an index for the tightness of the fit is common practice. Hence in summary, when executing linear regression we are presented with numerical values for $\hat{\beta}_0$, $\hat{\beta}_1$, MSE and R^2 .

Using the GLM Package

The basic Julia package that we use for carrying out linear regression is package `GLM`, standing for *Generalized Linear Models*. We describe the “generalized” notion in Section 9.3 below and for now use the package for nothing more than statistical inference for the model (8.14). In fact, we have already briefly used this package in Listing 8.3, lines 45–50 and in the ANOVA example of Section 7.3.

In Listing 8.5 we consider the `weightHeight.csv` dataset relating weights and heights of individuals. We carry out linear regression by invoking several alternative functions from `GLM`. These include `lm()` and the analogous use of `fit()` with a `LinearModel` argument. An alternative is to use `glm()` or the analogous use of `fit()` with a `GeneralizedLinearModel` argument together with `Normal()` and `IdentityLink()`. In all cases we specify a formula via

```
@formula(Height ~ Weight).
```

Such a Julia formula macro (marked by `@`) indicates that we are seeking a model where height (the dependent Y variable) is represented in terms of weight (the independent X variable). These are the names of the variables (columns) in the dataset. An alternative valid macro is

```
@formula(Height ~ Weight + 1).
```

Here the `+1` explicitly indicates to add an intercept term to the regression. However, by default it is not needed and is already assumed. If however you wish to carry out the regression without the intercept term then use `-1` instead.

As can be observed from the output, there are only very minor differences in output when using `lm()` vs. `glm()` (alternatively `fit()` with `LinearModel` vs. `GeneralizedLinearModel`). These have to do with the interpretation of the distribution of the test statistic for checking if $\hat{\beta}_i$ is significantly different from 0. More on that in the sequel. Essentially, when using `lm()` a T-distribution is used and when using `glm()` an normal distribution is used. This affects the p -value but not more.

The outcome of `lm()`, `glm()` or `fit()` is a model object that can then be used in various ways. A summary of the model can be printed using `println()` (or similar). Further, functions such as `deviance()`, `stderror()`, `dof_residual()`, `vcov()` and `r2()` and most importantly `coef()` can be applied to the model to obtain results from the regression. We use these functions in Listing 8.5 below. The listing also produces Figure 8.4. Notice that in this listing `lm1` and `lm2` are identical. Similarly, `glm1` and `glm2` are identical. We simply chose to present the `fit()` counterparts so that you see alternative ways to apply `lm()` and `glm()`.

Listing 8.5: Simple linear regression with GLM

```

1  using DataFrames, GLM, PyPlot, Statistics, CSV
2
3  data = CSV.read("weightHeight.csv")
4
5  lm1 = lm(@formula(Height ~ Weight), data)
6  lm2 = fit(LinearModel,@formula(Height ~ Weight), data)
7
8  glm1 = glm(@formula(Height ~ Weight), data, Normal(), IdentityLink())
9  glm2 = fit(GeneralizedLinearModel,@formula(Height ~ Weight), data, Normal(),
10            IdentityLink())
11
12 println("***Output of LM Model:")
13 println(lm1)
14 println("\n***Output of GLM Model:")
15 println(glm1)
16
17 pred(x) = coef(lm1)'*[1, x]
18
19 println("\n***Individual methods applied to model output:")
20 println("Deviance: ", deviance(lm1))
21 println("Standard error: ", stderror(lm1))
22 println("Degrees of freedom: ", dof_residual(lm1))
23 println("Covariance matrix: ", vcov(lm1))
24
25 yVals = data.Height
26 SStotal = sum((yVals .- mean(yVals)).^2)
27
28 println("R squared (calculated in two ways): ", r2(lm1),
29          ",\t", 1 - deviance(lm1)/SStotal)
30
31 println("MSE (calculated in two ways: ", deviance(lm1)/dof_residual(lm1),
32          ",\t", sum((pred.(data.Weight) - data.Height).^2)/(size(data)[1] - 2))
33
34 xlims = [minimum(data.Weight), maximum(data.Weight)]
35 plot(data.Weight, data.Height, "b.")
36 plot(xlims, pred.(xlims), "r")
37 xlim(xlims)
38 xlabel("Weight (kg)")
39 ylabel("Height (cm)");

```

```
***Output of LM Model:  
Formula: Height ~ 1 + Weight
```

Coefficients:

	Estimate	Std.Error	t value	Pr(> t)
(Intercept)	135.793	1.95553	69.4404	<1e-99
Weight	0.532299	0.0293556	18.1328	<1e-43

```
***Output of GLM Model:
```

```
Formula: Height ~ 1 + Weight
```

Coefficients:

	Estimate	Std.Error	z value	Pr(> z)
(Intercept)	135.793	1.95553	69.4404	<1e-99
Weight	0.532299	0.0293556	18.1328	<1e-72

```
***Individual methods applied to model output:
```

```
Deviance: 5854.057142765537  
Standard error: [1.95553, 0.0293556]  
Degrees of freedom: 198.0  
Covariance matrix: [3.8241 -0.0562853; -0.0562853 0.000861751]  
R squared (calculated in two ways): 0.6241443400847023, 0.6241443400847023  
MSE (calculated in two ways): 29.56594516548251, 29.56594516548251
```

- In line 3 we read the dataset. It is comprised of Height and Weight.
- In line 5-10 we create alternative models using package GLM as described above. In practice, you would only use one of these four alternative ways for creating the model.
- Lines 12-15 print the models (the lm() based model and the glm() based model). In both cases, the formula of the model is printed followed by a table that lists the coefficient estimates, followed by standard errors, test statistics (t-value or z-value for lm() or glm() respectively) and then p-values. We explain the meaning of these tables in the sequel.
- In line 17 we create the pred() function which uses the model to predict \hat{y} for a given x . It does this by taking the inner product of the coefficient vector obtained via coef() and the vector [1, x].
- Lines 19-32 present a variety of descriptors associated with the model. The function deviance() yields $SS_{\text{residuals}}$. The function stderror() yields standard error for the coefficient estimates (these are in agreement with the values in the tables). The function dof_residual() yields 198. This is the number of observations, 200 minus 2 as per the numerator of (8.17). The function vcov() yields the covariance matrix of the estimators as discussed in the sequel. We then present R^2 , both using the function r2() and via a manual calculation. We also present the MSE both using deviance() and via a manual calculation.
- Lines 34-39 produce Figure 8.4.

The Distribution of the Estimators

As the least squares estimators, $\hat{\beta}_0$ and $\hat{\beta}_1$, for the model (8.14) are random variables, we may compute their distribution. For this recall (8.7) and notice that the vector $\hat{\beta}$ is obtained via $\hat{\beta} = (A'A)^{-1}A'Y$. Combine this with (8.14) which by recalling (8.5) can be written as,

$$Y = A\beta + \epsilon,$$

to obtain,

$$\hat{\beta} = (A'A)^{-1}A'(A\beta + \epsilon) = \beta + (A'A)^{-1}A'\epsilon.$$

Now since ϵ is a zero mean Gaussian random vector we have that $\mathbb{E}[\hat{\beta}] = \beta$ or written element wise in the case of simple linear regression,

$$\mathbb{E}[\hat{\beta}_0] = \beta_0, \quad \mathbb{E}[\hat{\beta}_1] = \beta_1. \quad (8.18)$$

We thus see that the estimators are unbiased. Further to compute, the covariance matrix of the estimators consider, $\hat{\beta} - \beta = (A'A)^{-1}A'\epsilon$ and take the self outer product $(\hat{\beta} - \beta)(\hat{\beta} - \beta)'$ to get the matrix $(A'A)^{-1}A'\epsilon\epsilon'A(A'A)^{-1}$. The expectation of $\epsilon\epsilon'$ is $\sigma^2 I$ and hence the expectation of $(\hat{\beta} - \beta)(\hat{\beta} - \beta)'$ reduces to $\sigma^2(A'A)^{-1}$. This is the covariance matrix of $\hat{\beta}$. For the case of simple linear regression $A'A$ is a 2×2 matrix with an explicit inverse. It isn't hard to obtain,

$$\text{Cov}\left(\begin{bmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \end{bmatrix}\right) = \begin{bmatrix} \frac{\sigma^2 \sum_{i=1}^n x_i^2}{nS_{xx}} & -\frac{\bar{x}\sigma^2}{S_{xx}} \\ -\frac{\bar{x}\sigma^2}{S_{xx}} & \frac{\sigma^2}{S_{xx}} \end{bmatrix}, \quad \text{with} \quad S_{xx} = \sum_{i=1}^n (x_i - \bar{x})^2. \quad (8.19)$$

Under the normality assumption $\hat{\beta}$ is itself a Gaussian random vector with mean and covariance given by (8.18) and (8.19) respectively. We illustrate this in Listing 8.6 where we generate synthetic data according to (8.14) with $n = 10$ observations. We repeat this 10^4 times, each time obtaining new estimates $\hat{\beta}_0$ and $\hat{\beta}_1$. A cloud point of the estimators is plotted in Figure 8.5. It is plotted against a contour line of a normal distribution with the given mean and covariance matrix.

Listing 8.6: The distribution of the regression estimators

```

1  using DataFrames, GLM, PyPlot, Distributions, LinearAlgebra
2
3  beta0, betal = 2.0, 1.5
4  sigma = 2.5
5  n, N = 10, 10^4
6
7  function coefEst()
8      xVals = collect(1:n)
9      yVals = beta0 .+ betal*xVals + rand(Normal(0,sigma),n)
10     data = DataFrame([xVals,yVals],[::X,:Y])
11     model = lm(@formula(Y ~ X), data)
12     coef(model)
13 end
14
15 ests = [coefEst() for _ in 1:N]
16
17 plot(first.(ests),last.(ests),"b",ms="0.5")
18 plot(beta0,betal,"r")
19
20 xBar = mean(1:n)

```

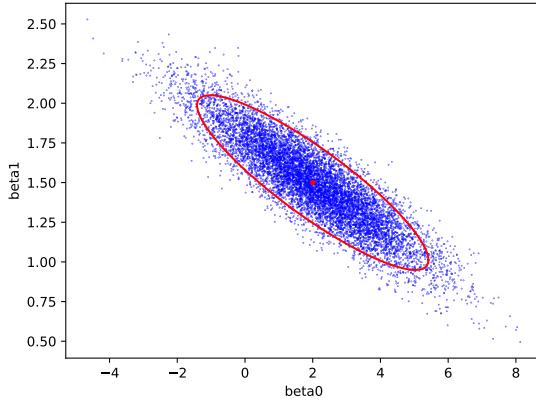


Figure 8.5: An illustration of the distribution of the estimators $\hat{\beta}_0$ and $\hat{\beta}_1$.

```

21 sXX = sum([(x - xBar)^2 for x in 1:n])
22 sx2 = sum([x^2 for x in 1:n])
23 var0 = sigma^2 * sx2/(n*sXX)
24 var1 = sigma^2/sXX
25 cv = -sigma^2*xBar/sXX
26
27 mu = [beta0, beta1]
28 Sigma = [var0 cv; cv var1]
29
30 r = 2.0
31 A = cholesky(Sigma).L
32 pts = [r*A*[cos(t),sin(t)] + mu for t in 0:0.01:2pi];
33
34 plot(first.(pts),last.(pts),"r")
35 xlabel("beta0")
36 ylabel("beta1");

```

- In lines 3-5 we set the parameters of this simulation. The assumed values are $\beta_0 = 2.0$, $\beta_1 = 1.5$ and $\sigma^2 = 2.5^2$.
- The function `coefEst()` in lines 7-13 generate a sequence x_1, \dots, x_n on $1, 2, \dots, n$ and then for these values generate y_1, \dots, y_n according to (8.14) in line 9. These values are then set in a `DataFrame`, and a linear model is generated. The return value is the vector `coef(model)`.
- In line 15 we create an array, `ests` of coefficients values, repeating N times.
- Lines 20-28 construct the mean vector `mu` and covariance matrix `Sigma` according to (8.18) and (8.19) respectively.
- Lines 30-32 plot a contour plot of the associated distribution.

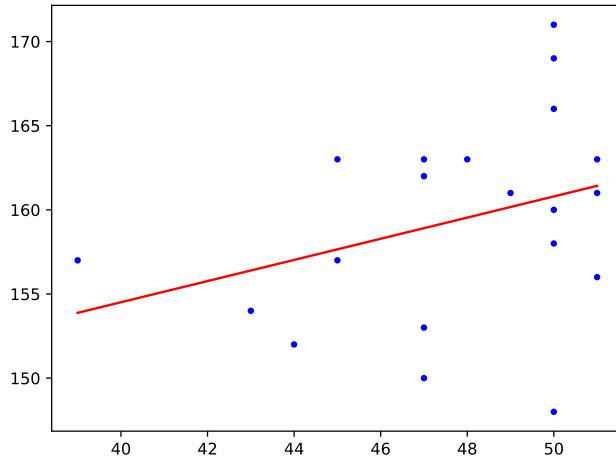


Figure 8.6: A scatter plot of Height vs. Weight with a line of best fit obtained via linear regression. Notice that while the line appears to be sloping up, the p -value is actually 0.1546 indicating there isn't a significant relationship of weight on height.

Statistical Inference for Simple Linear Regression

Now that we understand the distribution of $\hat{\beta}_0$ and $\hat{\beta}_1$ we can make use of it for hypothesis tests associated with the regression line. The main hypothesis test for simple linear regression is to check:

$$H_0 : \beta_1 = 0, \quad H_1 : \beta_1 \neq 0. \quad (8.20)$$

Here H_0 implies that there is no effect of X on Y whereas H_1 implies that there is an effect. A similar, although less popular, hypothesis test may also be carried out for the intercept β_0 :

$$H_0 : \beta_0 = 0, \quad H_1 : \beta_0 \neq 0. \quad (8.21)$$

In both hypothesis tests (8.20) and (8.21) we use a test statistic of the form,

$$T_i = \frac{\hat{\beta}_i}{S_{\hat{\beta}_i}},$$

where $i = 1$ for (8.20) and $i = 0$ for (8.21). In each case, the estimate of the standard error for $\hat{\beta}_i$ differs. For $i = 1$ we have,

$$S_{\hat{\beta}_1} = \sqrt{\frac{\text{MSE}}{S_{xx}}}, \quad (8.22)$$

whereas for the intercept case, $i = 0$ we have,

$$S_{\hat{\beta}_0} = \sqrt{\text{MSE} \left(\frac{1}{n} + \frac{\bar{x}^2}{S_{xx}} \right)}.$$

Both cases, make use of the MSE as defined in (8.17). It now turns out that in both cases, under H_0 , the test statistic, T_i is distributed according to a T-distribution with $n - 2$ degrees of freedom. This

can now be used to test the hypothesis (8.20) and (8.21) in a similar manner to that presented in Chapter 7. Note that it is also possible to adapt the hypothesis to test for $H_0 : \beta_i = \delta$, $H_1 : \beta_i \neq \delta$, for any desired δ or to create one sided hypothesis tests. However in practice, the test (8.20) is most useful.

We carry out this hypothesis test in Listing 8.7 where we only consider the first 20 observations of the `weightHeight.csv` dataset. As presented in Figure 8.6, one may see the regression line and believe that there is a relationship between X and Y . However this isn't the case! In this case, the variability of the data is too strong and we are not able to reject H_0 of (8.20) under any significant confidence level. This is due to the p -value of 0.1546 which results from the $\hat{\beta}_1 = 0.628733$ and a standard error (8.22) of 0.423107.

Listing 8.7: Hypothesis tests for simple linear regression

```

1  using DataFrames, GLM, PyPlot, Distributions, CSV
2
3  data = CSV.read("weightHeight.csv")
4  sData = sort(data, :Weight)[1:20,:]
5
6  model = lm(@formula(Height ~ Weight), sData)
7  pred(x) = coef(model)'*[1, x]
8
9  plot(sData.Weight, sData.Height, "b.")
10
11 xlims = [minimum(sData.Weight), maximum(sData.Weight)]
12 plot(xlims, pred.(xlims), "r")
13
14 tStat = coef(model)[2]/stderror(model)[2]
15 n = size(sData)[1]
16 pVal = 2*ccdf(TDist(n-2),tStat)
17 println("Manual Pval: ", pVal)
18 println(model)

```

```

Manual Pval: 0.15458691273390412
Formula: Height ~ 1 + Weight
Coefficients:
            Estimate Std.Error t value Pr(>|t|)
(Intercept) 129.359   20.2252 6.39594    <1e-5
Weight       0.628733  0.423107 1.48599    0.1546

```

- In line 3 we read the dataset and then sort the data frame according to `:Weight` in line 4. We then keep the first 20 entries.
- Lines 6-12 create `model` and plot the regression line as in previous examples.
- In lines 14-16 we calculate the test statistic and its p -value manually. The result is then printed in line 17 and when the model is printed in line 18 we can see in the second line of the printed table that the same p -value is obtained.

Confidence Bands and Prediction Bands

After collecting data, having a predicted model, $y = \hat{\beta}_0 + \hat{\beta}_1 x$ is useful for determining a prediction $\hat{y}(x^*)$ for every independent variable value x^* . For example, with the `weightHeight.csv` data used in Listing 8.5, based on $n = 200$ observations, we approximately have $\hat{\beta}_0 = 135.8$ and $\hat{\beta}_1 = 0.53$. Hence if we then consider an individual weighing $87kg$ then based on this model, their predicted height is,

$$\hat{y}(87) = 135.8 + 0.53 \times 87 = 181.9cm.$$

Having such a prediction is useful, however we would also like to obtain uncertainty bounds around $\hat{y}(87)$. Further, if instead of just $x^* = 87$ we would use x^* over some interval, then we would like to obtain *uncertainty bands*.

For this, we need to differentiate between two possible meanings of $\hat{y}(87) = 181.9$ or any other $\hat{y}(x^*)$. One meaning is that according to the model, $181.9cm$ is the expected height of individuals with a weight of $87kg$. Another meaning is that $181.9cm$ is the predicted height of an arbitrary individual with a weight of $87kg$. In both the expected value and predicted value case, our best possible estimate is $\hat{y}(x^*)$. However, when we consider uncertainty bounds (or bands) the widths of these bands differs depending on if we consider the expected value or a predicted value. This is similar to the difference between confidence intervals and prediction intervals presented in Chapter 6.

When considering expected values, the formula for *confidence bands* is:

$$\hat{y}(x^*) \pm t_{n-2,1-\frac{\alpha}{2}} \sqrt{\text{MSE} \times \left(\frac{1}{n} + \frac{(x^* - \bar{x})^2}{S_{xx}} \right)}. \quad (8.23)$$

Further, when considering predicted values, the formula for *prediction bands* is:

$$\hat{y}(x^*) \pm t_{n-2,1-\frac{\alpha}{2}} \sqrt{\text{MSE} \times \left(1 + \frac{1}{n} + \frac{(x^* - \bar{x})^2}{S_{xx}} \right)}. \quad (8.24)$$

In both cases, $1 - \alpha$ is the confidence level and quantiles of a T-distribution with $n - 2$ degrees of freedom are used. However in the prediction interval case (8.24) there is an additional 1 term not appearing in (8.23).

We illustrate these bands in Figure 8.7 generated by Listing 8.8. As can be observed from the figure, prediction bands are wider than confidence bands. If you were wishing to use the model to predict the height of a specific individual based on their weight you would use the blue prediction bands for uncertainty quantification. However, if you wanted to get a feel for possible models that could have resulted, you would use the green confidence bands.

Listing 8.8: Confidence and prediction bands

```

1  using DataFrames, GLM, PyPlot, Distributions, CSV
2
3  data = CSV.read("weightHeight.csv")
4  n = size(data)[1]
5  model = fit(LinearModel, @formula(Height ~ Weight), data)
6
7  alpha = 0.1
8  tVal = quantile(TDist(n-2), 1-alpha/2)

```

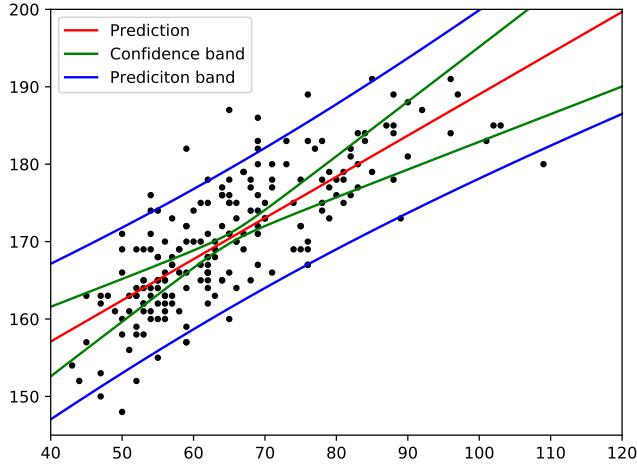


Figure 8.7: A scatter plot of Height vs. Weight with confidence bands (green) and predication bands (blue) along with a line of best fit (red).

```

9
10 xbar = mean(data$Weight)
11 Sxx = std(data$Weight)*(n-1)
12 MSE = deviance(model)/(n-2)
13
14 pred(x) = coef(model)'*[1, x]
15
16 interval(x, sign, prediction = 0) = sign(pred(x),
17             tVal * sqrt(MSE*(prediction+1/n+(x-xbar)^2/Sxx)) )
18
19 xGrid = 40:1:140
20 plot(data$Weight, data$Height, ".k")
21 plot(xGrid, pred(xGrid), "r", label="Linear model")
22 plot(xGrid, interval.(xGrid,+), "g", label="Confidence interval")
23 plot(xGrid, interval.(xGrid,-), "g")
24 plot(xGrid, interval.(xGrid,+,1), "b", label="Prediciton interval")
25 plot(xGrid, interval.(xGrid,-,1), "b")
26 xlim(40, 120)
27 ylim(145, 200)
28 legend(loc="upper left");

```

- In lines 3-5 we read the dataset and fit the model as in previous examples. We also set the number of observations, n .
- In lines 7-8 we set the significance level, α and the corresponding T-value.
- In lines 10-12 we calculate summary statistics, \bar{x} , S_{xx} and MSE.
- In line 14 we define the function `pred()`, determining $\hat{y}(x)$.
- In lines 15-16 we define the function `interval()` which is the main focus of this example. It is designed to implement both the upper bound and lower bound in (8.23) and (8.24). The argument x is x^* . The argument `sign` can literally be '+' or '-'. The argument `prediction`

has a default value of 0 indicating this is a confidence band as in (8.23). However if the value is set to 1 then equation (8.24) is obtained.

- The remainder of the code creates Figure 8.7. Observe the use of our `interval()` function applied via the broadcast `'.'` operator to `xGrid`. The second argument, `+` or `-` are actually the functions plus and minus respectively.

Checking Model Assumptions

Using a statistical model as in (8.14) allowed us to make a variety of conclusions about the population via statistical inference techniques. These include prediction, hypothesis testing and confidence bands as presented above. However, the validity of these techniques relies on the model assumptions of (8.14). This motivates us to check if *model assumptions* hold. The key model assumptions include:

Assumption I: A linear relationship between variables. More specifically $\mathbb{E}[Y | X = x]$ is a linear function in x .

Assumption II: Normally distributed errors around the regression line.

Assumption III: Equal variance for the errors around the regression line.

Assumption IV: Independent errors.

While a least squares line passing through a cloud of points is always a least squares line, if any of assumptions I-IV break, then the validity of the statistical results breaks. As a basic illustration, let us explore how things can go wrong with a classic adversarial example called *Anscombe's quartet*. This is a collection of four datasets, each with observations of the form $(x_1, y_1), \dots, (x_n, y_n)$. Anscombe's Quartet is useful in highlighting the dangers of applying a wrong model to the data. Although each of its four datasets have almost identical estimates for the regression line as well as for R^2 and other descriptors, the nature of each underlying dataset is vastly different. Hence if one was to blindly rely on descriptive statistics to gain an understanding of the four datasets, one would be misled without realising it. This becomes obvious once the datasets are visualized as in Figure 8.8.

Anscombe's quartet is presented in Listing 8.9 below. In this example the Anscombe's quartet is loaded from the `RDatasets` package, and a linear model solved for each of its datasets. The resulting four models are then plotted against their underlying data points in Figure 8.8, and the results show that although the coefficients of each model are the same, the nature of the underlying data is vastly different.

Listing 8.9: The Anscombe quartet datasets

```

1  using RDatasets, DataFrames, GLM, PyPlot
2
3  df = dataset("datasets", "Anscombe")
4
5  model1 = lm(@formula(Y1 ~ X1), df)
6  model2 = lm(@formula(Y2 ~ X2), df)

```

```

7  model3 = lm(@formula(Y3 ~ X3), df)
8  model4 = lm(@formula(Y4 ~ X4), df)
9
10 yHat(model, X) = coef(model)' * [ 1 , X ]
11 xlims = [0, 20]
12
13 subplot(221)
14 plot(df.X1, df.Y1,".b")
15 plot(xlims, [yHat(model1, i) for i in xlims], "r")
16 xlim(xlims)
17
18 subplot(222)
19 plot(df.X2, df.Y2,".b")
20 plot(xlims, [yHat(model2, i) for i in xlims], "r")
21 xlim(xlims)
22
23 subplot(223)
24 plot(df.X3, df.Y3,".b")
25 plot(xlims, [yHat(model3, i) for i in xlims], "r")
26 xlim(xlims)
27
28 subplot(224)
29 plot(df.X4, df.Y4,".b")
30 plot(xlims, [yHat(model4, i) for i in xlims], "r")
31 xlim(xlims)
32
33 println("Model 1. Coefficients: ", coef(model1)," \t R squared: ", r2(model1))
34 println("Model 2. Coefficients: ", coef(model2)," \t R squared: ", r2(model2))
35 println("Model 3. Coefficients: ", coef(model3)," \t R squared: ", r2(model3))
36 println("Model 4. Coefficients: ", coef(model4)," \t R squared: ", r2(model4))

```

Model 1. Coefficients: [3.00009, 0.500091]	R squared: 0.6665424595087749
Model 2. Coefficients: [3.00091, 0.5]	R squared: 0.6662420337274844
Model 3. Coefficients: [3.00245, 0.499727]	R squared: 0.6663240410665592
Model 4. Coefficients: [3.00173, 0.499909]	R squared: 0.6667072568984651

- In line 1 the `RDataSets` package is loaded.
- In line 3 Anscombe's quartet dataset is loaded from the `RDataSets` package via the `dataset()` function. This function takes two arguments, the name of the data package in `RDataSets` containing Anscombe's quartet ("datasets"), and the name of the dataset ("Anscombe"). The dataset is stored as the data frame `df`.
- In lines 5 to 8 a linear model for each of the four datasets is created. Note that `df` has 8 columns total, with the individual four datasets comprising of x - y pairs (e.g `X1, Y1`). Note that each model is a simple linear model of the form (8.14).
- In line 10 the function `yHat()` is created, which takes a model type as input, and a vector of predictor values `X`, and outputs a corresponding vector of predictions `yHat`. Note that it uses the coefficients of the model `coef()`, and the design matrix.
- In line 13 the data points for the first dataset are plotted, and in line 14 the corresponding linear model is plotted via the use of the `yHat` function.

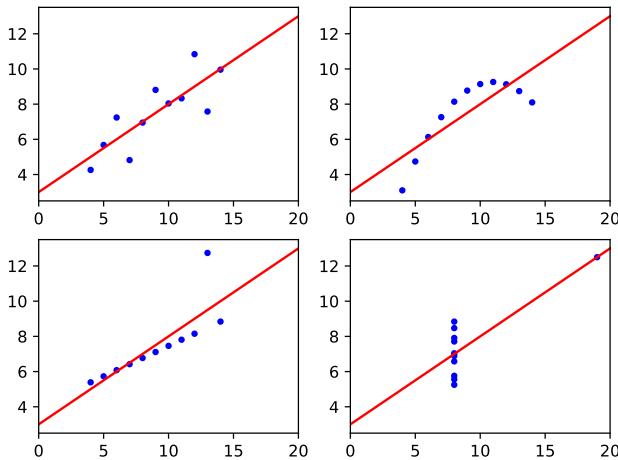


Figure 8.8: Plot of Anscombe's quartet. Although each dataset has nearly identical descriptive statistics, and that each result in almost identical linear models, it can be seen that the underlying datasets are very different.

- The remaining three datasets and corresponding linear models are plotted in the remaining lines.
- It can be seen from Figure 8.8 that even though the models are the same, the underlying datasets are very different.

In general, most accepted techniques for checking model assumptions involve considering the *residuals*. These are constructed by estimating \hat{y}_i for each value of x_i and then setting,

$$e_i = y_i - \hat{y}_i, \quad \text{for } i = 1, \dots, n. \quad (8.25)$$

The residuals were already presented in (8.4). Least squares minimizes their sum of squares. It can be shown from the normal equations that,

$$\sum_{i=1}^n e_i = 0,$$

and hence the arithmetic mean of the residuals is also 0. Analysis of model assumption then amounts to analyzing the way in which the residuals fluctuate around 0. This is often done via a *residual plot*, where all the residuals for a dataset are plotted. Such a visualization then allows to see if there is any strong indication that assumptions I, II, III or IV don't hold. If the residuals appear to oscillate around 0 then assumption I may not hold. If a normal probability plot of the residuals (see example below) does not exhibit a linear line then assumption II may not hold. If the spread of the residuals around 0 is not uniform the assumption III does not hold. Finally if there appear to be correlations between errors then assumption IV may not hold. For this one may also conduct a *Wald-Wolfowitz runs test*, however we omit the details.

In Listing 8.10 below, we construct a plot of the residuals of the data, along with a normal probability plot as described in Chapter 4. For this dataset, the residuals don't display any abnormal features that give a strong indication of violating assumptions I-IV.

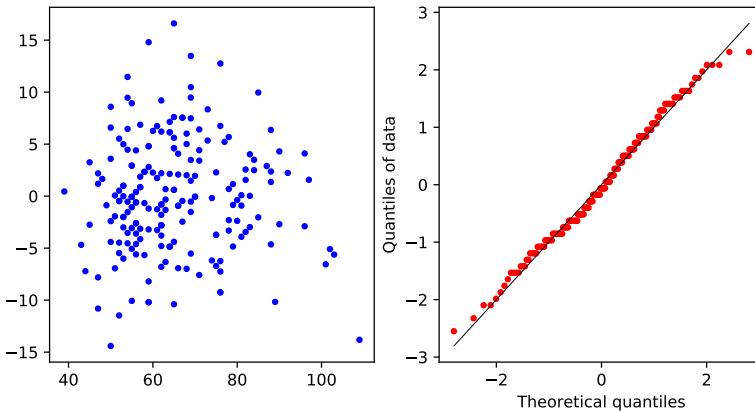


Figure 8.9: Plot of residuals of the data, and a plot of the theoretical quantiles against the actual quantiles of the data.

Listing 8.10: Plotting the residuals and their normal probability plot

```

1  using DataFrames, GLM, PyPlot, Distributions, CSV
2
3  function normalProbabilityPlot(data)
4      mu = mean(data)
5      sig = std(data)
6      n = length(data)
7      p = [(i-0.5)/n for i in 1:n]
8      x = quantile(Normal(),p)
9      y = sort([(i-mu)/sig for i in data])
10     plot(x, y, ".r")
11     xRange = maximum(x) - minimum(x)
12     plot([minimum(x), maximum(x)],
13           [minimum(x), maximum(x)], "k", lw=0.5)
14     xlabel("Theoretical quantiles")
15     ylabel("Quantiles of data")
16 end
17
18 data = CSV.read("weightHeight.csv")
19
20 model = lm(@formula(Height ~ Weight), data)
21 pred(x) = coef(model)'*[1, x]
22
23 residuals = data.Height - pred.(data.Weight)
24
25 figure(figsize=(8,4))
26 subplot(121)
27 plot(data.Weight, residuals,"b.")
28
29 subplot(122)
30 normalProbabilityPlot(data[:,3]);

```

- In lines 3 to 16 the function `normalProbabilityPlot()` is created. This function takes an array of response observations, and then calculates the ankit of the set. First the quantiles of the normal distributed are calculated based on the locations given by formula $k - 0.5/n$ (which divides the range $[0,1]$ into equally spaced bins).

- In line 9 a comprehension is used to normalize the observations, which are then sorted from smallest to largest via the `sort()` function.
- In line 10 the normalized sorted observations `y` are plotted against the theoretical quantiles expected of a normal observation `x`.
- In line 18 the data is loaded as the data frame `data`, and in line 19 only the female data is selected. Note the use of the `.==` to select only the rows which correspond to female (1), and all columns are selected by `:`.
- In line 23 the residuals are calculated based on the implementation of (8.25). Note the use of `pred()` function along with `'.'`.
- In lines 25 to 30 Figure 8.9 is created. In line 27 the residual plot is created, while in line 30 the `normalProbabilityPlot` function is used to create the normal probability plot.
- The results show that for the residual plot, most residuals are scattered evenly around zero, while in the normal probability plot the quantiles of the data roughly match the theoretically expected quantiles.

8.3 Multiple Linear Regression

Having looked at linear regression involving one variable, in this section we look at situations involving linear regression of more than one variable. We now generalize the model (8.14) by extending from a single independent variable to p independent variables:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p + \epsilon_i. \quad (8.26)$$

Here, ϵ_i is still a single normally distributed random variable for every i . The data for this model involves n tuples of the form: $(y_1, x_{11}, \dots, x_{1p}), \dots, (y_n, x_{n1}, \dots, x_{np})$ where $n > p$. Each such tuple is an observation with a dependent variable y_i and independent variables x_{i1}, \dots, x_{ip} . In this case, the least squares estimation of (8.4) carries over in a straightforward manner. This is now with an $n \times (p+1)$ design matrix A , generalizing (8.5) to

$$A = \begin{bmatrix} 1 & x_1 & \dots & x_{1p} \\ 1 & x_2 & \dots & x_{2p} \\ \vdots & \vdots & & \vdots \\ 1 & x_n & \dots & x_{np} \end{bmatrix}. \quad (8.27)$$

The least squares results of Section 8.1 and many of the statistical analysis results of Section 8.2 carry over from the case of $p = 2$ to general p . We now explore an example and then consider some aspects found in multiple linear regression that aren't exhibited in simple linear regression.

A Multiple Linear Regression Example

First we cover the case of linear regression where all the variables are continuous in nature. For our example, the `cpus` dataset from `RDataSets MASS` package is used. This dataset is a record of the relative performance measured of several old CPU's, along with their respective attributes, such as clock speed, cache size, etc.

In Listing 8.11 linear regression is used to fit a model that predicts the performance of a CPU based on several characteristics. In creating this regression, we believed that a linear relationship can hold for the “CPU frequency” but not for its reciprocal, the “clock cycle speed”. For this we created a new variable, `Freq` by transforming the original variable, `CycT`. Such *variable transformations* are often common in regression analysis as one often seeks to transform the data so that a linear model like (8.26) is applicable.

As you may see from the output of the listing, there are now $p + 1$ parameter estimates (5 in our case). Each estimate has its own standard error, T-value and a resulting p -value, similar to the simple linear regression case. Such a display of the variables often allows to get an immediate feel for the quality of the regression model. For example in our case, we see that all variables exhibit extremely low p -values with the exception of `Freq`. Even though this variable still appears meaningful for the model.

With such a model at hand we compare two hypothetical computers, A and B. Computer A has a smaller cache, but is 100 times faster in terms of frequency. The predicted values for the two computers are presented and we see that computer B is expected to have slightly better performance according to this model.

Listing 8.11: Multiple linear regression

```

1  using RDataSets, GLM, Statistics
2
3  df = dataset("MASS", "cpus")
4  df.Freq = map( x->10^9/x , df.CycT)
5
6  model = lm(@formula(Perf ~ MMax + Cach + ChMax + Freq), df)
7  pred(x) = round(coef(model) * vcat(1,x), digits = 3)
8
9  println("n = ", size(df)[1])
10 println("(Avg,Std) of observed performance: ", (mean(df.Perf), std(df.Perf)))
11 println(model)
12 println("Estimated performance for computer A: ", pred([32000, 32, 32, 4*10^7]))
13 println("Estimated performance for computer B: ", pred([32000, 16, 32, 6*10^7]))
```

```

n = 209
(Avg,Std) of observed performance: (105.61722488038278, 160.8305871990777)
```

```
Formula: Perf ~ 1 + MMax + Cach + ChMax + Freq
```

Coefficients:

	Estimate	Std.Error	t value	Pr(> t)
(Intercept)	-46.5763	7.62382	-6.10931	<1e-8
MMax	0.00841457	0.000639214	13.1639	<1e-28
Cach	0.872508	0.152825	5.70919	<1e-7
ChMax	0.96736	0.234847	4.11911	<1e-4

```

Freq      9.74951e-7  5.5502e-7  1.75661  0.0805
Estimated performance for computer A: 320.564
Estimated performance for computer B: 326.103

```

- In line 2 the "cpus" dataset from the "MASS" RDatasets package is stored as the data frame df.
- In line 3 the cycle time CyCT is used to calculate the frequency via the map function. Since the cycle time is in nanoseconds, one can calculate the cycles per second via $10^9/x$. These frequency values are then appended to the data frame df as Freq.
- In line 6 the model is created, where the response variable Perf is the published performance (relative to an IBM 370/158-3), and the response variables are: MMax-the maximum main memory in kilobytes (KB), Cach-the cache size in KB, ChMax-the maximum number of channels, and Freq-the frequency in cycles per second.
- In line 8 the details of the model are printed. From the results it can be seen that the p -value for each of the coefficients, except for Freq, are all less than 0.05, and are therefore significant for $\alpha = 0.05$.
- In lines 9 and 10 the coefficients of the model are used to estimate the performance of a computer with the specified attributes shown.

Collinearity

When conducting multiple linear regression it is possible for some subset of the explanatory variables, X_1, \dots, X_p , to be dependent. This situation is called *collinearity* or *multicollinearity*. In extreme situations this may be due to redundancy of the data or multiple readings. Imagine for example that temperature readings are present both in the Centigrade scale as one variable and the Fahrenheit scale in another variable.

In other situations, collinearity is present due to inherent statistical relationships between variables. For example, assume we are trying to predict the salary of individuals, Y . For this we consider the age of individuals, X_1 and the years of experience, X_2 . In this case if we ignore career interruptions then,

$$X_2 = X_1 - D,$$

where D , a *latent variable*, is the age of the individual at which she started employment. This immediately renders X_1 and X_2 to be dependent random variables. Such a dependence may be very strong if the variability of D isn't large.

Perfect collinearity renders the design matrix A , (8.27) to have less than $p + 1$ independent columns (*less than full rank*). In such a case, the matrix $(A'A)$ is not invertible and the pseudo-inverse cannot be computed as in (8.6) or (8.8). Still there are ways around this problem via the singular value decomposition as in(8.9), or other means that we present below. However, in many cases collinearity isn't perfect and while algebraic problems don't exist, numerical and statistical problems still persist.

A consequence of collinearity is a breakdown of the model assumptions of the linear regression model. This typically does not affect the least squares estimates, however it does imply that the model is extremely sensitive to perturbations of the data. It also means that p -values and other statistical summaries from the model are distorted. There are several suggested ways for detecting the presence of collinearity. Some involve considering R^2 values, others involve attempting to regress explanatory variables via others, and another way involves considering the covariances between variables or the covariance between variable estimates.

Listing 8.12 presents an artificial example with X_1, X_2 and X_3 , where,

$$X_3 = X_1 + 2X_2 + \text{noise}. \quad (8.28)$$

When the variance of the noise is significant, collinearity isn't highly present in the example. However as the variance of the noise decreases, the linear relationship involving X_1, X_2 and X_3 creates significant collinearity.

Listing 8.12: Exploring collinearity

```

1  using Distributions, GLM, DataFrames, PyPlot, Random, LinearAlgebra
2
3  n = 100
4  beta0, beta1, beta2, beta3 = 10, 30, 60, 90
5  sig = 25
6  sigX = 5
7  etaVals = [50.0, 10.0, 1.0, 0.1, 0.001, 0.0]
8
9  function createDataFrame(eta)
10    Random.seed!(1)
11    x1 = round.(collect(1:n) + sigX*rand(Normal(),n), digits = 5)
12    x2 = round.(collect(1:n) + sigX*rand(Normal(),n), digits = 5)
13    x3 = round.(x1 + 2*x2 + eta*rand(Normal(),n), digits = 5)
14    y = beta0 .+ beta1*x1 + beta2*x2 + beta3*x3 + sig*rand(Normal(),n)
15    return DataFrame(Y = y, X1 = x1, X2 = x2, X3 = x3)
16  end
17
18  for eta in etaVals
19    println("\n **** eta = $(eta):")
20    df = createDataFrame(eta)
21    glmOK = true
22    try
23      global model = lm(@formula(Y ~ X1 + X2 + X3), df)
24    catch err
25      println("\nException with GLM: ", err, "!!!!\n\n")
26      glmOK = false
27    end
28
29    if glmOK
30      covMat = vcov(model)
31      sigVec = sqrt.(diag(covMat))
32      corrrmat = round.(covMat ./ (sigVec*sigVec'), digits=6)
33      println("Cov(X1,X3) = ", corrrmat[2,4], "\t Cov(X2,X3) = ", corrrmat[3,4])
34      println(model)
35    else
36      A = [ones(n) df.X1 df.X2 df.X3]
37      psInv(lambda) = inv(A'*A + lambda*I)*A'
38      for lam in [1000, 1, 0.5, 0.1, 0.01, 0.001, 0.0001, 0.0]
39        println("lam = ", lam,

```

```

40           "\t coeff:", psInv(lam)*df.Y,
41           "\t pInv diff: ", round(norm(psInv(lam)-pinv(A)), digits=6))
42       end
43   end
44 end

```

```

***** eta = 50.0:
Cov(X1,X3) = -0.267706, Cov(X2,X3) = -0.164598
            Estimate Std.Error t value Pr(>|t|)
(Intercept) 6.81369  5.42102 1.2569  0.2118
X1          29.7057  0.419327 70.8415 <1e-83
X2          60.3347  0.395592 152.517 <1e-99
X3          89.993  0.0505589 1779.96 <1e-99

***** eta = 10.0:
Cov(X1,X3) = -0.614248, Cov(X2,X3) = -0.769126
            Estimate Std.Error t value Pr(>|t|)
(Intercept) 6.81369  5.42102 1.2569  0.2118
X1          29.7339  0.511995 58.0745 <1e-75
X2          60.391  0.610544 98.9135 <1e-97
X3          89.9648  0.252795 355.881 <1e-99

***** eta = 1.0:
Cov(X1,X3) = -0.988048, Cov(X2,X3) = -0.996993
            Estimate Std.Error t value Pr(>|t|)
(Intercept) 6.81369  5.42102 1.2569  0.2118
X1          30.0503  2.62097 11.4653 <1e-18
X2          61.024  5.03503 12.1199 <1e-20
X3          89.6483  2.52795 35.4629 <1e-56

***** eta = 0.1:
Cov(X1,X3) = -0.999873, Cov(X2,X3) = -0.99997
            Estimate Std.Error t value Pr(>|t|)
(Intercept) 6.81356  5.42101 1.25688 0.2118
X1          33.2174  25.3443 1.31064 0.1931
X2          67.358  50.5243 1.33318 0.1856
X3          86.4813  25.2794 3.42102 0.0009

***** eta = 0.001:
Cov(X1,X3) = -1.0, Cov(X2,X3) = -1.0
            Estimate Std.Error t value Pr(>|t|)
(Intercept) 6.81519  5.42137 1.2571  0.2118
X1          378.839  2529.18 0.149787 0.8812
X2          758.601  5058.2  0.149975 0.8811
X3          -259.14  2529.12 -0.102463 0.9186

***** eta = 0.0:

```

Exception with GLM: PosDefException(4)!!!!

lam = 1000.0	coeff:[0.465989, 23.4209, 38.0522, 99.5252]	pInv diff: 0.192246
lam = 1.0	coeff:[6.77793, 19.6425, 40.2071, 100.057]	pInv diff: 0.007371
lam = 0.5	coeff:[6.90282, 19.6379, 40.2094, 100.057]	pInv diff: 0.003756
lam = 0.1	coeff:[7.00622, 19.6341, 40.2114, 100.057]	pInv diff: 0.000763
lam = 0.01	coeff:[7.02993, 19.6333, 40.2118, 100.057]	pInv diff: 7.7e-5
lam = 0.001	coeff:[7.03231, 19.6332, 40.2118, 100.057]	pInv diff: 8.0e-6
lam = 0.0001	coeff:[7.03255, 19.633, 40.2117, 100.057]	pInv diff: 1.0e-6
lam = 0.0	coeff:[7.03257, 108.537, 236.278, 63.0257]	pInv diff: 0.018495

- In lines 3-7 we define the basic parameters of this experiment: `n` is the number of observations; the `beta` variables are the actual β_i values used to generate the data; `sig` is the standard deviation of the error term; `sigX` determines variability on the x -values; and `etaVals` determines a range of values for the variability of the noise in (8.28).
- In lines 9-16 we define our function `createDataFrame()`. It creates data based on `eta` determining the variability of the noise in (8.28).
- Lines 18-44 iterate over `etaVals`, each time trying to fit a linear model in line 23. Here we use Julia's try-catch mechanism to catch an exception in case `lm()` throws an exception. This happens in the case of `eta = 0.0` in which case the matrix $A'A$ is singular. Then in lines 30-34 we output covariance values and GLM output when an exception isn't thrown. However in lines 36-41, we attempt ridge regression (see next section) in case of an exception.

8.4 Model Adaptations

Transformations of Variables to Make a Linear Model

We now introduce the idea of transforming variables as a means of solving models which are not linear in nature. To that end we present an example where linear regression is used to solve a model with a single predictor, but where the response is polynomial in nature (of degree 2). In this case we wish to run linear regression on a model of the following form,

$$Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \epsilon. \quad (8.29)$$

Here, the model is only dependent on one variable X , however the third term is polynomial in nature, i.e. X^2 . Hence we cannot directly use a linear model to find the values of the coefficients. In order to overcome this we make use of the concept of the *transformation of variables* to transform (8.29) into a linear combination of random variables.

The approach here involves simply denoting the existing random variables as new random variable. By letting $X_1 = X$ and $X_2 = X^2$, we can transform (8.29) into,

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \epsilon \quad (8.30)$$

Here X_1 is defined as the values in column X , while X_2 is defined as the square of the values in column X . Now with the linear equation (8.30), we can create a linear model, and solve for the coefficients β_0 , β_1 , and β_2 .

In Listing 8.13 below, we first load the original data, then append an additional column `x2` to our data frame. The values in this appended column `x2` take on the square of the values stored in column `x`. Then the `lm()` function is used to solve the linear model of (8.30), and the resulting solution plotted along with the original data points in Figure 8.10.

Listing 8.13: Linear regression of a polynomial model

```

1  using DataFrames, GLM, PyPlot, CSV
2
3  data = CSV.read("polynomialData.csv")

```

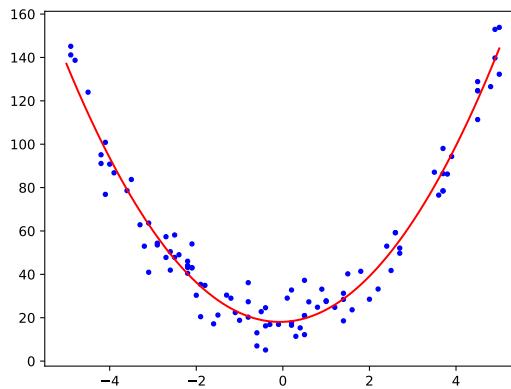


Figure 8.10: Plot of the linear regression polynomial model, alongside the original data points.

```

4  data.X2 = abs2.(data.X)
5  model = lm(@formula(Y ~ X + X2), data)
6
7  xGrid = -5:0.1:5
8  yHat(x) = coef(model)[3]*x.^2 + coef(model)[2]*x + coef(model)[1]
9  plot(data.X, data.Y, ".b")
10 plot(xGrid, yHat.(xGrid), "r");
11 println(model)

```

```

Formula: Y ~ 1 + X + X2

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 18.0946   1.16533 15.5274 <1e-27
X           0.705362  0.298831 2.36041  0.0203
X2          4.90144   0.108245 45.281   <1e-66

```

- In line 3 the data from the csv file is loaded and stored as the data frame `data`.
- In line 4 a second column, `data[:X2]`, is appended to the data frame `data`. The values are defined as the squares of the values in column `X` through the use of the `abs2()` and `'.'`.
- In line 5 the linear model (8.30) is implemented via the `@formula` function, and this model is then solved via `lm()` and stored as `model`.
- In line 8 the function `yHat(x)` is created which makes model predictions for a given value of `x` based on the coefficients of the solved model `model`.
- In lines 9 and 10, the data in `data` is plotted alongside model predictions made via `yHat`. It can be seen from the resulting output that the predicted model is polynomial in nature and is a good fit of the data.
- In line 11 the underlying model is printed, and it can be seen that the coefficients of the predicted model, β_0 , β_1 and β_2 , are all significant with p -values less than 0.05.

Discrete and Categorical Variables

We now cover linear regression in the context of discrete/categorical variables. As discussed previously in Section 4.1, while a continuous variable can take on any value on a continuous domain (such as temperature, length, and mass), a *categorical variable* is a variable that only takes on discrete values. Importantly, the discrete values, groups, or *levels* of a categorical variable are not ordered in nature. Examples of categorical variables include sex (male/female), and specific colors (e.g. Red, Green, Blue). As a side point, an *ordinal variable* is similar to a categorical variable in the fact that observations fall into discrete groups, however an ordinal variable implies that some ordering of the levels exists, such as exam gradings A, B, C, or ratings such as low, medium, high.

To provide further insight we now present an example of regression involving two variables, a categorical variable and a continuous variable. However, before we do so it is important to realize there are two different ways of constructing a model when there is more than one predictor variable present. The first way is to assume that the predictor variable only has an additive effect on the model (i.e. the variables do not interact with each other). The second way is to assume that an *interaction effect* exists between the variables. The concept of interaction effects are discussed in the subsequent subsection, but for now we consider that no interaction effect exists.

In the case of linear regression involving a single continuous variable X and a categorical variable with n levels with no interaction effects present, the model can be represented as follows,

$$Y = (\beta_0 + \beta_2 \mathbf{1}_2 + \cdots + \beta_n \mathbf{1}_n) + \beta_1 X + \epsilon. \quad (8.31)$$

Here, the indicator function for each level $\mathbf{1}_2, \dots, \mathbf{1}_n$ only takes on a value of 1 for that level, and is zero when considering any other level. That is, for the first (default) level, all indicators are zero, and the coefficients β_0 and β_1 are the models intercept and slope. For the second level, $\mathbf{1}_2 = 1$ while all other indicators are zero, hence the coefficients $\beta_0 + \beta_2$ and β_1 are the models intercept and slope. One can see that the terms $\beta_2 \mathbf{1}_2 + \cdots + \beta_n \mathbf{1}_n$ encapsulate the additional additive effects that each subsequent level has on the model.

In Listing 8.5 below we provide an example based on the `weightHeight.csv` dataset. Previously this dataset was used in Listing 8.5, where a simple linear model of height based on weight was created, with the variable `sex` excluded from the model. We now repeat the same process, and consider the categorical variable `sex` as part of our model. Importantly, this is done based on the assumption that no interaction effect exists between the variables `sex` and `weight`.

Listing 8.14: Regression with categorical variables - no interaction effects

```

1  using CSV, RDatasets, DataFrames, GLM, PyPlot
2
3  df = CSV.read("weightHeight.csv")
4  mW = df[df.Sex == "M", :Weight]
5  mH = df[df.Sex == "M", :Height]
6  fW = df[df.Sex == "F", :Weight]
7  fH = df[df.Sex == "F", :Height]
8  categorical!(df, :Sex)
9  model = lm(@formula(Height ~ Weight + Sex), df)
10
11 predFemale(x) = coef(model)[1:2]'*[1, x]
12 predMale(x)   = [sum(coef(model)[[1,3]]) coef(model)[2]]*[1, x]
13

```

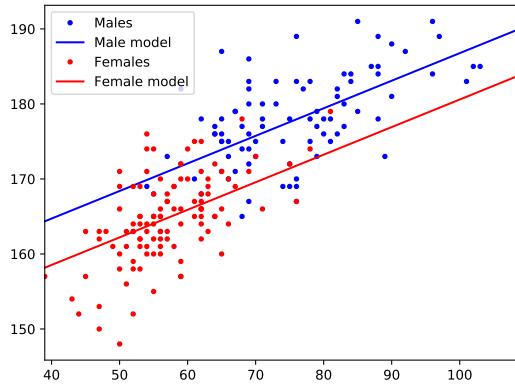


Figure 8.11: Linear model with categorical (sex) variable, with no interaction effect.

```

14 xlims = [minimum(df[:Weight]), maximum(df[:Weight])]
15
16 plot(mW, mH, "b.", label="Males")
17 plot(xlims, predMale.(xlims),"b", label="Male model")
18
19 plot(fW, fH, "r.", label="Females")
20 plot(xlims, predFemale.(xlims),"r", label="Female model")
21 xlim(xlims);
22 legend(loc="upper left")
23 println(model)

```

Formula: Height ~ 1 + Weight + Sex

Coefficients:

	Estimate	Std.Error	t value	Pr(> t)
(Intercept)	143.834	2.21522	64.9298	<1e-99
Weight	0.367539	0.0378728	9.70456	<1e-17
Sex: M	6.18277	0.999302	6.18709	<1e-8

- In lines 3 to 7 the data is loaded as the data frame df, and then the data frame is further sliced into four arrays containing the weights and heights of males and females respectively. In line 4 all rows containing male data is selected via df[:Sex] .== "M", and then the corresponding weights stored as the array mW. This same logic is repeated in rows 5 to 7.
- In line 8 the categorical! () function is used to change the :Sex column of the DataFrame df to the Categorical type.
- In line 9 the linear model is created based on the formula Height ~ Weight + Sex. Note the use of the + operator, which represents an additive effect only, and not an interaction effect between the variables.
- In line 11 the function predFemale is created, which uses the coefficients of the model to predict the height based on a given weight. Note that the design matrix is multiplied by the first two coefficients of the model, selected via [1:2], which correspond to the intercept and

slope for the first (i.e. default) level of the `Sex` variable. Note that the first level of the `Sex` variable is determined by the value of the first row (i.e. F).

- In line 12 the function `predMale` is created, in a similar manner to line 11. However, in this case the design matrix is multiplied by the sum of the first and third coefficients of the model, and the second coefficient of the model. Here the first coefficient represents the intercept for females, and the third coefficient represents the additive effect that the sex `male` has on the model (note the significance of `Sex`: M in the models output coefficients).
- In lines 16 to 23 the data is plotted along with the models for both males and females. Since that the model is based on no interaction effect between variables, the slope of the model for males remains the same.

Understanding Interactions

Having looked at the case of linear regression involving a continuous and categorical variable with no interaction effect, we now cover the concept of an *interaction effect*. An interaction effect is when two variables interact, that is, when the value of one variable directly changes the effect of another.

In the case of linear regression involving a single continuous variable X and a categorical variable with n levels with an interaction effect present, the model can be represented as follows,

$$Y = (\beta_0 + \beta_2 \mathbf{1}_2 \dots \beta_{2n} \mathbf{1}_n) + (\beta_1 + \beta_3 \mathbf{1}_2 \dots \beta_{2n-1} \mathbf{1}_n)X + \epsilon. \quad (8.32)$$

Here, the indicator function for each level $\mathbf{1}_2, \dots, \mathbf{1}_n$ only takes on a value of 1 for that level, and is zero for all others (in the same manner as in (8.32)). One will notice the formulas are somewhat similar, however here the coefficient of X depends on the level considered. For example, in the case of the first (i.e. default) level, the model is,

$$Y_1 = \beta_0 + \beta_1 X + \epsilon,$$

while for the second level is

$$Y_2 = (\beta_0 + \beta_2) + (\beta_1 + \beta_3)X + \epsilon.$$

Hence the choice of level contributes to both the intercept and slope terms of the linear model. We now provide an example, where we revisit the `weightHeight.csv` dataset.

In Listing 8.15 below we replicate Listing 8.5, however rather than assuming an additive effect between the `Sex` and `Weight` variables, we include an interaction effect between them instead.

Listing 8.15: Regression with categorical variables - with interaction effects

```

1  using CSV, RDatasets, DataFrames, GLM, PyPlot
2
3  df = CSV.read("weightHeight.csv")
4  mW = df[df.Sex == "M", :Weight]
5  mH = df[df.Sex == "M", :Height]
6  fW = df[df.Sex == "F", :Weight]
7  fH = df[df.Sex == "F", :Height]
8  categorical!(df, :Sex)

```

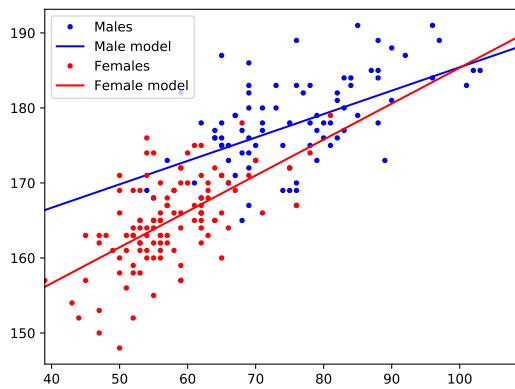


Figure 8.12: Linear model with categorical (sex) variable, with interaction effect.

```

9  model = lm(@formula(Height ~ Weight * Sex), df)
10
11 predFemale(x) = coef(model)[1:2]'*[1, x]
12 predMale(x)   = [sum(coef(model)[[1,3]]) sum(coef(model)[[2,4]]])*[1, x]
13
14 xlims = [minimum(df[:Weight]), maximum(df[:Weight])]
15
16 plot(mW, mH, "b.", label="Males")
17 plot(xlims, predMale.(xlims),"b", label="Male model")
18
19 plot(fW, fH, "r.", label="Females")
20 plot(xlims, predFemale.(xlims),"r", label="Female model")
21 xlim(xlims);
22 legend(loc="upper left")
23 println(model)

```

```

Formula: Height ~ 1 + Weight + Sex + Weight & Sex

Coefficients:
Estimate Std.Error t value Pr(>|t|)
(Intercept) 137.469 3.75357 36.6234 <1e-88
Weight 0.478912 0.0651662 7.34908 <1e-11
Sex: M 16.7398 5.14467 3.25382 0.0013
Weight & Sex: M -0.166745 0.0797367 -2.09119 0.0378

```

- This listing is identical to Listing 8.5, except for line 9.
- In line 9 the linear model is created based on the formula `Height ~ Weight * Sex`. Note the use of the `*` operator, which represents an interaction effect between the variables.
- In line 11 the function `predFemale` is created which predicts height based on weight in the same manner as line 11 of Listing 8.5.
- In line 12 the function `predMale` is created in a similar manner to that of line 12 of Listing 8.5. Note that since the second level (i.e. male) of the `Sex` variable is considered here, the intercept coefficient is given by the sum of the first and third coefficients of the model, while the slope

coefficient is given by the second and fourth coefficients of the model. Note that the third and fourth coefficients of the model, `Sex:M` and `Weight&Sex:M`, correspond to β_2 and β_3 of (8.32) respectively.

- From the model output and Figure 8.12 it can be seen that the categorical variable `sex` has an statistically significant effect on the prediction of height based on weight, and that the p -value for each coefficient is less than 0.05.

Simpson's Paradox

Having covered categorical variables and interaction effects, we now investigate the so-called *Simpson's paradox*, or *Yule-Simpson effect*. This paradox, which sometimes occurs in statistics, is the observation that a trend present in the data can disappear or reverse when the data is divided into subgroups. Although simple in intuition, it is an important concept to remember, as one must always be careful when constructing a model, as there may be another hidden factor within the data that may significantly change the results and conclusions.

In Listing 8.16 below an example of Simpson's paradox is presented. In it, the `IQalc.csv` dataset is used, which contains measurements of individuals IQ's, along with a rating of their weekly alcohol consumption, for three separate groups A,B and C. From this dataset a linear model predicting alcohol consumption based on IQ is first made, with the individual groups not taken into account. Another model is then created, with each group treated separately, and with an interaction effects taken into account (the same as in Listing 8.15). The resulting Figure 8.13 illustrates Simpson's paradox, as the first model suggests that people with higher IQ's drink more, however when the individual groups are taken into account, this trend is reversed, suggesting people with higher IQ's drink less.

Listing 8.16: Simpson's paradox

```

1  using DataFrames, GLM, PyPlot, CSV
2
3  df = CSV.read("IQalc.csv")
4  groupA = df[df.Group .== "A", :]
5  groupB = df[df.Group .== "B", :]
6  groupC = df[df.Group .== "C", :]
7
8  model  = fit(LinearModel, @formula(AlcConsumption ~ IQ), df)
9  modelA = fit(LinearModel, @formula(AlcConsumption ~ IQ), groupA)
10 modelB = fit(LinearModel, @formula(AlcConsumption ~ IQ), groupB)
11 modelC = fit(LinearModel, @formula(AlcConsumption ~ IQ), groupC)
12
13 pred(x)  = coef(model)' * [1, x]
14 predA(x) = coef(modelA)' * [1, x]
15 predB(x) = coef(modelB)' * [1, x]
16 predC(x) = coef(modelC)' * [1, x]
17
18 xlims = collect(extrema(df.IQ))
19
20 fig = figure(figsize=(10, 5))
21 subplot(121)
22 plot(df.IQ, df.AlcConsumption, "b.", alpha=0.1)
23 plot(xlims, pred.(xlims), "b", label="All data")

```

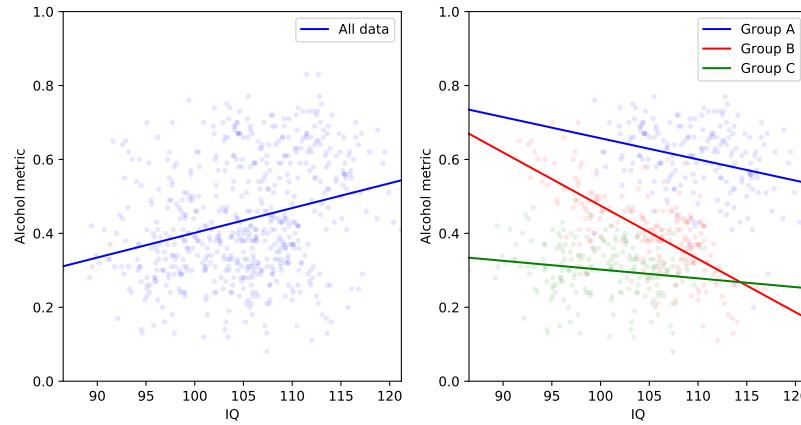


Figure 8.13: An illustration of Simpson’s paradox. The trend in the data reverses when the additional variable (group) is taken into account.

```

24 xlim(xlims), ylim(0,1)
25 xlabel("IQ")
26 ylabel("Alcohol metric")
27 legend(loc="upper right")
28
29 subplot(122)
30 plot(groupA.IQ, groupA.AlcConsumption, "b.", alpha=0.1)
31 plot(groupB.IQ, groupB.AlcConsumption, "r.", alpha=0.1)
32 plot(groupC.IQ, groupC.AlcConsumption, "g.", alpha=0.1)
33 plot(xlims, predA.(xlims), "b", label="Group A")
34 plot(xlims, predB.(xlims), "r", label="Group B")
35 plot(xlims, predC.(xlims), "g", label="Group C")
36 xlim(xlims), ylim(0,1)
37 xlabel("IQ")
38 ylabel("Alcohol metric")
39 legend(loc="upper right")

```

Formula: AlcConsumption ~ 1 + IQ + Group + IQ & Group

Coefficients:

	Estimate	Std.Error	t value	Pr(> t)
(Intercept)	1.22983	0.13369	9.19914	<1e-18
IQ	-0.00572488	0.00121418	-4.71502	<1e-5
Group: B	0.686308	0.175165	3.91807	<1e-4
Group: C	-0.690499	0.179828	-3.83978	0.0001
IQ & Group: B	-0.00868815	0.00162265	-5.35431	<1e-6
IQ & Group: C	0.00335154	0.001708	1.96226	0.0502

- In lines 3 to 10 the data is loaded. In line 4 the `df [:Group` column is classified as a categorical variable via the `CategoricalArray()` function. In lines 5 to 10 the data for the individual groups IQ’s and alcohol consumptions are stored into separate arrays as shown.
- In line 12 a linear model is created based on all of the data available. Note that the categorical variable `Group` is not part of this model.

- In line 13 the function `pred` is created which predicts alcohol consumption based on the model of line 12.
- In line 15 a linear model is created based on the `IQ` and `Group` variables. Note that an interaction effect between the two variables is included via the use of the `*` operator in the models formula.
- In lines 16 to 19 the functions `predA`, `predB` and `predC` are created. These three functions predict alcohol consumption for each of the three levels of the `Group` variable. Note that in each case, and in particular lines 17 and 18, the intercept and slope is a sum of the first levels coefficients, and the subsequent component that each group has on both the intercept and slope.
- The remaining lines are used to generate Figure 8.13. The results show that when the individual groups are taken into account, the trend reverses, showing that alcohol consumption is inversely related to `IQ`.

8.5 Model Selection

Note: This section is still under construction.

The concept of *model selection* deals with selecting the best model from a set of possible models. Although this can involve aspects of experimental design, here we focus purely within the scope of creating the best model from a given dataset.

Importantly, since many possible models exist, the decision to accept a model is based not only on accuracy, but also on the models complexity. That is, if two models have roughly the same statistical power, then the simpler model is typically chosen. This is because in such cases it is generally more likely that the simpler model would be correct, as an over-complicated model may lead to issues such as over-fitting (see Figure 8.1 for motivation).

There are many methods in statistics and machine learning for model selection, none of which are accepted as the universally ideal method. In most cases, model selection is cyclic in nature, and the general approach is summarized in the four steps below. The process involves starting at step one and repeating steps 1. to 4. until, based on some policy, a suitable model is reached.

1. Choose a model and fit it to the data.
2. Make predictions based on the model or measure quality using means such as p -values.
3. Compare the model predictions to the data.
4. Use this information to update the model and repeat.

In Listing 8.17 an example of model selection is performed via *stepwise regression*. The method adopted here involves solving a model, eliminating the least significant variable, re-solving the model, and repeating this process until a final model is reached. For this example, the starting model is the same as that of Listing 8.11. Once solved, the p -values of the coefficients are compared against

the specified threshold, and if the largest (i.e. least significant) value is greater than this threshold, the corresponding least significant variable is eliminated. The model is then solved again, and the process repeated until the p -values of all coefficients are less than the specified threshold. In this trivial example only one variable is eliminated, however for a dataset of hundreds of variables, the importance of model selection and the method of stepwise regression becomes obvious.

Listing 8.17: Basic model selection

```

1  using RDatasets, GLM, PyPlot
2
3  df = dataset("MASS", "cpus")
4  df.Freq = map( x->10^9/x , df.CycT)
5  df = df[:, [:Perf, :MMax, :Cach, :ChMax, :Freq]]
6
7  function stepReg(df, reVar, pThresh)
8      predVars = setdiff(names(df), [reVar])
9      fm = Formula(reVar, Expr(:call, :+, predVars...))
10     model = lm(fm, df)
11     pVals = [p.v for p in coeftable(model).cols[4]]
12
13     while maximum(pVals) > pThresh
14         deleteat!(predVars, findmax(pVals)[2]-1)
15         fm = Formula(reVar, Expr(:call, :+, predVars...))
16         model = lm(fm, df)
17         pVals = [p.v for p in coeftable(model).cols[4]]
18     end
19     model
20 end
21
22 model = stepReg(df,:Perf, 0.05)
23 println(model)
24 println("Estimated performance for a specific computer (after model reduction):",
25         coef(model)'*[1, 32000, 32, 32])

```

Formula: Perf ~ 1 + MMax + Cach + ChMax

Coefficients:

	Estimate	Std.Error	t value	Pr(> t)
(Intercept)	-40.938	6.95029	-5.89011	<1e-7
MMax	0.00905833	0.000526383	17.2086	<1e-40
Cach	0.945043	0.147888	6.39027	<1e-8
ChMax	0.869351	0.229281	3.79165	0.0002

Estimated performance for a specific computer (after model reduction): 306.9891801461281

- In lines 3 and 4 the data is loaded and the Freq variable calculated and appended to the data frame df in the same manner as those of Listing 8.11.
- In line 5 the variables Perf, MMax, Cach, ChMax and Freq of the data frame are selected, and stored as df.
- In lines 7 to 20 the function stepReg() is created. This function performs stepwise linear regression for a given data frame df, based on a specified response variable from the data frame reVar (as a symbol), and a maximum p -value specified pThresh. It first calculates a linear model based on all the variables in df, and then compares the p -values of the predictor

variables against pThresh and removes the largest one if it is greater than pThresh. This process is repeated until all the models coefficient p -values are less than pThresh.

- In line 8 the predictor variables are selected and stored as an array of symbols predVars. Here the response variable reVar is removed from the list of variables through the use of the names() and setdiff() functions.
- In line 9 (8.26) is implemented via the Formula() function. Formula() takes two arguments, the response variable as a symbol, and the second is a Julia expression that describes the right hand side of the equation. The expression is created via Expr(), within which :call is used to call the operation +, and performs this on the remaining arguments. Through the use of the splat operator ... the call is performed over all elements of predVars..
- In line 10 the model is solved based on the formula fm for the data in the data framedf.
- In line 11 the p -values of the model are extracted as an array through the use of a comprehension. First the coefstable() function is used and then the fourth column selected (i.e. the column of p -values). Finally the v field (i.e. value field) of each of these elements is selected via the comprehension. The resulting p -values are stored as pVal.
- In lines 13 to 18 a while() condition is used to perform stepwise regression, where the largest p -value is eliminated each iteration. This terminates when the largest p -value is greater than pThresh.
- In line 14 the index of the variable with the largest p -value is calculated via indmax , and then the deleteat! function used to remove this variable form the array of predictor variables predVars.
- In line 15 the model formula is created via fm in the same way as in line 9, but based on the updated array predVars from the previous line.
- In line 16 the newly updated model is solved via the lm function.
- In line 17 the p -values are selected in the same way as in line 11.
- In line 18 the model is then solved again.
- Once the condition in line 13 is no longer satisfied, the while loop terminates and returns the model in line 19.
- In line 22 the function steReg is used to perform stepwise regression on the data frame df, in order to predict the variable Perf, given a significance threshold of 0.05.
- In line 23 the model output is printed. It can be seen that stepwise regression has eliminated the variable Freq.
- In lines 24 and 25 the model is used to predict the performance for a specific computer, based on the same attributes of the example on line 10 of Listing 8.11. Note that in this example the Freq property has been excluded since this is not a parameter of our model. The resulting estimate of the 306.99 is roughly in agreement with the estimate of 320.56 from Listing 8.11. Although the elimination of one variable is trivial, in cases with many variables, stepwise regression proves to be a useful model simplifying tool.

Chapter 9

Machine Learning Basics - DRAFT

Note: This chapter is still under construction. Nevertheless, you may make use of some of the material as is.

We now shift focus from classical statistics to *machine learning*. In doing so we explore *classification* and *regression* in the context of *supervised learning*. We also briefly explore methods of *unsupervised learning*, *reinforcement learning* and *generational models*.

In Section 9.1 we consider the basic setup of supervised learning. In Section 9.2 we explore bias and variance as well as regularization. Section 9.3 we briefly explore generalized linear models including logistic regression. In Section 9.4 we explore some further supervised machine learning techniques including decision trees, support vector machines and deep neural networks. We close with Section 9.5 where we explore some unsupervised learning techniques including clustering and Principle Component Analysis (PCA). We close with Section 9.6 we briefly explore Markov decision processes and reinforcement learning. Then in Section 9.7 we explore *Generative Adversarial Networks (GANs)*.

9.1 Training, Validation and Testing

This section is under construction.

The general data setup of supervised learning is comprised of n observations. Each observation $i \in \{1, \dots, n\}$ consists of a pair (x_i, y_i) where $\{x_i\}_{i=1}^n$ is generally called the *data* and $\{y_i\}_{i=1}^n$ are the *labels*. Each data point x_i is often a non-small or even high dimensional vector representing recorded voice, images or even video. Labels are often taken from a smaller set of values.

When presented with data comprised of x and y , the *classification problem* is to create a model say, \hat{f} which can map data to labels. If the labels are assumed to take on a continuum of values then the task is sometimes called a *regression problem*, however we focus on classification problems here where the labels are either $\{0, 1\}$ or falling in some other small set, such as $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

The general assumption is that the dataset presents a good representation of a broader (typically

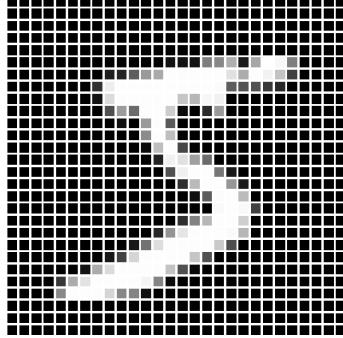


Figure 9.1: The first image of the MNIST dataset. Each image in MNIST is 28×28 , i.e. containing 784 grayscale pixels.

infinite) collection of $(\tilde{x}_i, \tilde{y}_i)$ pairs that still haven't been encountered. Once a model is learned, it can be used in production by being applied to new data points and treating $\hat{y}_i = \hat{f}(\tilde{x}_i)$ as a predicted value. For example, the examples in this section use the popular *MNIST digits dataset* (Modified National Institute of Standards and Technology database) where each x_i is an image of a digit as in Figure 9.1 and the corresponding label, y_i , records the actual digit that the image represents. Creating a model $\hat{f}(\cdot)$ for such a dataset implies training (or learning) a function, $\hat{f}(\cdot)$ that generally adheres to the following two general objectives:

$$\hat{f}(x_i) = y_i, \quad \text{for as many } i \text{ as possible,} \quad (9.1)$$

$$\hat{y}_i = \hat{f}(\tilde{x}_i) = \tilde{y}_i, \quad \text{as often as possible.} \quad (9.2)$$

A problem with such a formulation is that the first objective (9.1) is fully attainable by encoding the dataset exactly in \hat{f} and forcing $\hat{f}(x_i) = y_i$ to hold for all i . Doing so, would generally be at the cost of the second objective (9.2). This is called *overfitting*. Hence we generally seek a model \hat{f} that aims towards the first objective while keeping \hat{f} not too complicated and doesn't overfit (see also Figure 8.1 for this idea in the context of regression).

Since evaluating the second objective is not possible as it deals with unknown data, we often split our data into a *training set* and a *test set*. For example, the standard MNIST example has $n = 60,000$ images treated as a training set and an additional 10,000 images treated as a test set. See Figure 9.1 for an example image. We may even further split out a chunk of the training set and call it the *validation set*. The validation set can be used to tune model parameters, prior to testing. Other related methods include *cross validation*.

Once the training set is obtained, our job is to fit a model to the training set while making sure to avoid overfitting. One clear way of avoiding overfitting is to use simple models. Another is to keep evaluating model fit using the validation set. In any case, once the test set is evaluated we can detect if we overfit the model or not by comparing performance.

9.2 Bias, Variance and Regularization

This section is under construction.

Regularization

A key concept often applied in modern statistics and machine learning is *regularization*. The main idea is to take the data fitting objective as in (8.3),

$$\min_{\beta} L(\text{data}, \beta), \quad (9.3)$$

and augment the loss function $L(\cdot, \cdot)$, with a regularization term that depends on a *regularization parameter* λ :

$$\min_{\beta} L(\text{data}, \beta) + R(\lambda, \beta). \quad (9.4)$$

Now λ , often a scalar in the range $[0, \infty)$ but also sometimes a vector, is a *hyper parameter* that allows to regulate the problem and help with problems such as: collinearity, overfitting and model selection. A common general regularization technique is *elastic net* where $\lambda = [\lambda_1 \ \lambda_2]'$ and,

$$R(\lambda, \beta) = \lambda_1 \|\beta\|_1 + \lambda_2 \|\beta\|^2.$$

Here $\|\beta\|^2 = \sum_{i=1}^p \beta_i^2$ and $\|\beta\|_1 = \sum_{i=1}^p |\beta_i|$. Hence the values of λ_1 and λ_2 determine what kind of penalty the objective function will pay for high values of β_i . Clearly with $\lambda_1, \lambda_2 = 0$ the original objective isn't changed. Further as $\lambda_1, \lambda_2 \rightarrow \infty$ the estimates $\beta_i \rightarrow 0$ and the data is fully ignored. The virtue of regularization is that there is often a magical “midway” λ where the objective (9.4) does a much better job than the unregularized (9.3).

Particular cases of elastic net are the more classic *ridge regression* (also called *Tikhonov regularization*) and *LASSO* standing for *least absolute shrinkage and selection operator*. In the former $\lambda_1 = 0$ and only λ_2 is used and in the latter $\lambda_2 = 0$ and only λ_1 is used. One of the virtues of LASSO (also present in the more general elastic net case) is that the $\|\beta\|_1$ cost allows to knock out variables by “zeroing out” their β_i values. Hence LASSO is very useful as an advanced model selection technique, especially when the number of variables, p is very large (it can even be that $p > n$ yielding the problem to be categorized as *high dimensional*). We don't deal with LASSO explicitly but refer the reader to explore the Julia packages `GLMNet.jl` and `Lasso.jl`. See also Section C for further information.

The case of ridge regression is slightly simpler to analyze. In this case the data fitting problem can be represented as,

$$\min_{\beta \in \mathbb{R}^{p+1}} \|A\beta - y\|^2 + \lambda \|\beta\|^2,$$

where we now consider λ as a scalar (previously λ_2) in the range $[0, \infty)$. The squared norm, $\|\cdot\|^2$ of a vector is just its inner product with itself and A is the $n \times (p+1)$ design matrix (8.27) comprised of x values and y . The problem can just be rephrased as

$$\min_{\beta \in \mathbb{R}^{p+1}} \|(A + \lambda I)\beta - y\|^2.$$

And is thus solved by applying the pseudo-inverse of $A + \lambda I$ to the vector y . Incidentally, for any $\lambda > 0$ it can be shown that $A + \lambda I$ has linearly independent columns. This even holds if A has some dependent columns (and isn't full rank). Thus perfect collinearity is alleviated by adding a regularization term. It can even be shown that A^\dagger of (8.9) satisfies,

$$A^\dagger = \lim_{\lambda \rightarrow 0} ((A + \lambda I)'(A + \lambda I))^{-1}(A + \lambda I)'.$$

Similarity (and much more practically), collinearity that isn't perfect, can also be alleviated by considering non-zero λ values. This type of *shrinkage estimator* is very popular.

Listing 9.1 presents an example of ridge regression where we carry out *k-fold cross validation* to find a good λ value.

Listing 9.1: Ridge regression with *k*-fold cross validation

```

1  using RDatasets, DataFrames, Random, LinearAlgebra, MultivariateStats
2  Random.seed!(0)
3
4  df = dataset("MASS", "cpus")
5  n = size(df)[1]
6  df = df[shuffle(1:n), :]
7
8  K = 10
9  nG = Int(floor(n/K))
10 n = K*nG
11 println("Loosing $(size(df)[1] - n) observations.")
12
13 lamMin, lamMax = 0.0, 1.0
14 lamVals = collect(lamMin:(lamMax-lamMin)/(K-1):lamMax)
15
16 testSet(k) = collect(1+nG*(k-1):nG*k)
17 trainSet(k) = setdiff(1:n,testSet(k))
18
19 yTest(k) = convert(Array{Float64,1},df[testSet(k),:Perf])
20 yTrain(k) = convert(Array{Float64,1},df[trainSet(k),:Perf])
21
22 xTest(k) = convert(Array{Float64,2},df[testSet(k),[:Cach, :ChMin]])
23 xTrain(k) = convert(Array{Float64,2},df[trainSet(k),[:Cach, :ChMin]])
24
25 betas = [ridge(xTrain(k),yTrain(k),lamVals[k]) for k in 1:K]
26 errs = [norm([ones(nG) xTest(k)]*betas[k] - yTest(k)) for k in 1:K]
27 bestLambda = lamVals[findmin(errs)[2]]
28
29 macro RR(x) return:(round.($x,digits = 2)) end
30
31 println("Tried lambdas: ", @RR lamVals)
32 println("Errors: ", @RR errs)
33 println("Found best lambda for regularization: ", bestLambda)
34
35 betaFinal = ridge(convert(Array{Float64,2},df[:,[:MMin, :Cach, :ChMin]]),
36                      convert(Array{Float64,1},df[:,:Perf]),bestLambda)
37
38 println("Beta estimate: ", betaFinal)
```

```

n = 209, K = 10. Loosing 9 observations.
Tried lambdas: [0.0, 0.11, 0.22, 0.33, 0.44, 0.56, 0.67, 0.78, 0.89, 1.0]
Errors: [726.12, 614.98, 687.93, 1366.78, 1423.93, 1073.51, 1212.32, 1145.91, 714.84, 726.81]
Found best lambda for regularization: 0.1111111111111111
Beta estimate: [0.0236763, 1.04624, 3.7681, -6.36191]
```

- In lines 4-6 we read the data frame, get the number of observations n and then shuffle the observations.

- In lines 8-11 we determine variables associated with the cross validation. The variable `K` determines the number of groups and then `nG` determines the number of observations per group. We then reassign a value to `n` as the number of effective observations. We print out the number of observations that are lost as a remainder.
- In lines 13-14 we determine the range of λ values for ridge regression. The array, `lamVals` is the set of values that is tested via cross validation.
- In lines 16-17 we define the functions `testSet()` and `trainSet()` for determining the sets of indices for training and testing in the k -fold cross validation. These functions for the y variables and x variables of both testing and training are defined in lines 19-20 and lines 22-23 respectively.
- Line 25 uses the `ridge()` function the `MultivariateStats` package to carry out ridge regression for each train set `k` using the associated value of λ from `lamVals[k]`.
- In line 26, we calculate the prediction error of each such λ value, each time with the corresponding set.
- The in line 27 we use the second return value of `findmin()` to find the minimal index of a λ value and extract that value from the array `lamVals`. This is the `bestLambda`.
- In line 29 we define a simple macro for rounding output. Then the remainder of the code outputs our `Beta estimate` and we see that the best λ for regularization was $\lambda = 0.111$.

9.3 Logistic Regression and the Generalized Linear Model

This section is still under construction.

Up to now we dealt with statistical models of the form $Y = \beta'X + \epsilon$ (where we consider the first element of the vector X to be 1 allowing for a β_0 term). These models are linear because the dependence of Y on X is linear. Still we were able to transform elements of X , as in the polynomial example (8.29) to accommodate for some non-linear relationships as long as transformed values of X interact linearly. Since X is considered non-random in the regression, such transformations allowed us to stay in the realm of linear regression and least squares. However, what if we wanted to transform Y ? This is where *Generalized Linear Models (GLM)* come into the picture.

For a GLM, we choose a one-to-one real function $g(\cdot)$ and call it the *link function*. We then set,

$$g(Y) = \beta'X + \epsilon, \quad \text{or} \quad Y = g^{-1}(\beta'X + \epsilon),$$

where $g^{-1}(\cdot)$ is the *inverse link function*. Now remembering from (8.1) that in regression we wish to consider the conditional expectation of Y given $X = x$, we have,

$$\hat{y}(x) = \mathbb{E}[g^{-1}(\beta'x + \epsilon)].$$

The random component in this expectation is ϵ and for any distribution of ϵ and every link function there is some expected value function $\hat{y}(x)$. In this linear regression case, the link function is just the identity function $g(y) = y$ in which case $\hat{y}(x) = \beta'x$. This is because expectation is linear

and ϵ is zero mean. However, now in the generalized linear model, the expected value is generally more complicated. In the $g(y) = y$ case and assuming ϵ is normally distributed, $\hat{\beta}$ found via least squares was also the MLE. This was discussed briefly when we presented the log-likelihood (8.15). However, in the GLM case, least squares does not generally yield the MLE. We rather need to use other numerical methods.

It turns out that when ϵ follows a distribution from an *exponential family* there is a corresponding suitable link function, $g(\cdot)$ that allow finding the MLE of β using efficient algorithms. The exponential family of distributions is actually a “clan” of distributions that encompasses the normal distribution, the exponential distribution, the Poisson distribution and many more common distributions. We don’t discuss it further in the book, however its relevance in GLM is interesting. The practical point is that there are distribution - link function pairs that allow for simple modeling and efficient procedures for finding the MLE $\hat{\beta}$. When we fit a GLM model to data, Julia’s GLM package, searches for the MLE. It then uses asymptotic properties of MLE for yielding standard errors and p -values associated with each $\hat{\beta}_i$.

We first focus on one of the most common GLM models, *logistic regression*. We then present an additional example, using GLM for prediction.

Logistic Regression

Sometimes a dataset may consist of predictor variables which are continuous, but where the recorded outcomes are binary in nature, such as heads/tails, success/failure or 0/1. In these cases, it makes no sense to use a linear model, since the output can only take on two possible values. Instead, *logistic regression* can be used instead to calculate the parameters of a *logistic model*.

In a logistic model, the outcomes are represented by the indicator function, with positive (i.e. successful) outcomes considered 1, and negative outcomes considered 0. The logistic model then considers that the log of the odds is a linear combination of the predictors, as shown below,

$$\log \left(\frac{Y}{1-Y} \right) = \beta_0 + \beta_1 X_1 + \dots + \beta_p X_p + \epsilon \quad \text{where} \quad Y = \mathbb{P}(\text{success}). \quad (9.5)$$

Note that the left hand side of this equation is also known as the *logit function*. Considering (9.5) as a GLM we have that,

$$g(y) = \frac{y}{1-y} \quad \text{and} \quad g^{-1}(u) = \frac{1}{1+e^{-u}} = \frac{e^u}{1+e^u}.$$

The inverse of the logit function is called the *sigmoid function* also known as a *logistic function*. One suitable distribution for ϵ is a Bernoulli distribution. In this case we have that $\hat{y}(x) = g^{-1}(\beta x)$

In the case of a single predictor variable, this can be re-arranged as,

$$\hat{y} = \frac{1}{1+e^{-(\beta_0+\beta_1 x)}}. \quad (9.6)$$

In Listing 9.2 below, an example is presented based on the results of an exam for a group of students, with only one predictor variable. In this example, the result for each student has been

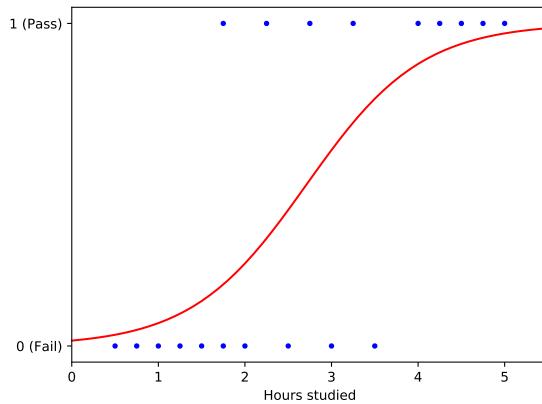


Figure 9.2: Logistic regression performed on the resulting exam data of students.

recorded as either pass (1), or fail (0), along with the number of hours each student studied for the exam. In this example the GLM package is used to perform logistic regression. The resulting model is plotted in Figure 9.2, and the associated model and coefficients printed below.

Listing 9.2: Logistic regression

```

1  using GLM, DataFrames, Distributions, PyPlot, CSV
2
3  data = CSV.read("examData.csv")
4
5  model = glm(@formula(Pass ~ Hours), data, Binomial(), LogitLink())
6
7  pred(x) = 1/(1+exp(-(coef(model)[1] + coef(model)[2]*x)))
8
9  xGrid = 0:0.1:maximum(data[:Hours])
10
11 plot(data.Hours, data.Pass, "b.")
12 plot(xGrid, pred.(xGrid), "r")
13 xlabel("Hours studied")
14 xlim(0, maximum(data[:Hours]))
15 yticks([0,1], ["0 (Fail)", "1 (Pass)"]);

```

Formula: Pass ~ 1 + Hours

Coefficients:

	Estimate	Std.Error	z value	Pr(> z)
(Intercept)	-4.07771	1.76098	-2.31559	0.0206
Hours	1.50465	0.628716	2.3932	0.0167

- In line 3 the data is loaded into the data frame `data`.
- In line 5 the `glm()` function is used to create a logistic model based from the data in `dataframe`, and stores the model as `model`. The `@formula()` macro is used to create a formula as the first argument, where the predictor variable is the `hours` column, and the response the `Pass` column. The data frame `data` is specified as the second argument. The third argument `t` is the type of family from which the data comes from, specified here as

`Binomial()`, and the fourth argument is the type of link function used, specified here as `LogitLink()`.

- In line 7 the coefficients of the model are stored as `C`.
- In lines 8 to 15 Figure 9.2 is plotted, the main calculations of which are done in lines 8 and 9. In line 8 a `linspace` is created over which the logistic model is plotted.
- In line 9 equation (9.6) is implemented over the domain created in line 8. Note the use of the coefficients of the model, i.e. the elements of `C`, used in the calculation.
- The output of the model can be seen below Figure 9.2. Note that β_0 corresponds to the intercept estimate, while β_1 corresponds to the hours estimate. The *p*-value of both of these coefficients can be seen to be < 0.05 and hence are statistically significant.

Other GLM Examples

Listing 9.3 explores several alternative GLM models.

Listing 9.3: Exploring generalized linear models

```

1  using GLM, RDatasets, DataFrames, Distributions, PyPlot, Random, LinearAlgebra
2  Random.seed!(0)
3
4  df = dataset("MASS", "cpus")
5  n = size(df)[1]
6  df = df[shuffle(1:n), :]
7
8  pTrain = 0.2
9  lastTindex = Int(floor(n*(1-pTrain)))
10 numTest = n - lastTindex
11
12 train = df[1:lastTindex,:]
13 test = df[lastTindex+1:n,:]
14
15 formula = @formula(Perf ~ CycT + MMin + MMax + Cach + ChMin + ChMax)
16 model1 = glm(formula, train, Normal(), IdentityLink())
17 model2 = glm(formula, train, Poisson(), LogLink())
18 model3 = glm(formula, train, Gamma(), InverseLink())
19
20 invIdentityLink(x) = x
21 invLogLink(x) = exp(x)
22 invInverseLink(x) = 1/x
23
24 A = [ones(numTest) test.CycT test.MMin test.MMax test.Cach test.ChMin test.ChMax]
25 pred1 = invIdentityLink.(A*coef(model1))
26 pred2 = invLogLink.(A*coef(model2))
27 pred3 = invInverseLink.(A*coef(model3))
28
29 actual = test[:Perf]
30 lossModel1 = norm(pred1 - actual)
31 lossModel2 = norm(pred2 - actual)
32 lossModel3 = norm(pred3 - actual)
33
34 println("Model 1: ", coef(model1))
```

```

35  println("Model 2: ", coef(model2))
36  println("Model 3: ", coef(model3))
37  println("\nLoss of models 1,2,3: ",(lossModel1 ,lossModel2, lossModel3))

Model 1: [-60.3184, 0.055669, 0.0175752, 0.00435287, 0.907027, -1.76895, 2.36445]
Model 2: [3.91799, -0.00161407, 9.77454e-6, 3.1481e-5, 0.00577684, 0.00519083, 0.00156646]
Model 3: [0.00993825, 6.12058e-5, 1.42618e-7, -1.5675e-7, -2.71825e-5, -7.61944e-5, 1.2299e-5]

Loss of models 1,2,3: (558.944925722054, 360.11867318943433, 577.4165274822029)

```

- In lines 4-6 we setup the data frame based on the cpus dataset. We randomly shuffle the rows in line 6.
- Lines 8-10 determine the indices of the training set and test set. The training set data frame, `train`, is then determined in line 12. The test set data frame, `test` is determined in line 13.
- Line 15 sets the `formula` to be used for GLM. The dependent variable is `Perf` and the independent variables are `CyCT`, `MMin`, `MMax`, `Cach`, `ChMin` and `ChMax`.
- Lines 16-18 create three `glm` models: `model1` is a standard linear model; `model2` has a `LogLink()` link function with `Poisson()` error; and `model3` has an `InverseLink()` link function with `Gamma()` error.
- In lines 20-22 we define the inverse link functions.
- The design matrix for the test data, `test` is constructed in line 24.
- In lines 25-27, predictions for the test data are constructed.
- In line 29 the actual performance of the test is obtained.
- The performance of models 1, 2 and 3 is tested in lines 30-32.
- The remainder of the code prints the results.

9.4 Supervised Learning Methods

In the preceding sections we covered various aspects of linear regression and generalized linear regression. These supervised learning methods allow us to derive models based on *labelled data*. The response (i.e. label) of each observation of predictor variables (i.e. observation of several *features*) is known. However, linear regression and GLM are only one aspect of multivariate analysis and machine learning. We now explore other methods, popular with *machine learning* and the study of *high dimensional data*. Such methods have proved useful across a wide variety of problems and domains ranging from optimization and classification, to prediction based problems, and others. Machine learning and many of its aspects, are widely used in practice, such as in recommender systems, image processing, spam and content filtering, and many more domains.

There are infinite possibilities for models \hat{f} , yet within machine learning a few notable classes have emerged. In this section we briefly explore four types:

Linear least squares classifiers - These methods use least squares as covered in previous sections together with threshold functions. They allow to create simple classifiers.

Support Vector Machines (SVM) - These methods use separating hyperplanes to create classifiers.

Decision trees and random forest - These methods create decision trees for classifying data. Random forest is a bagging algorithm applied to decision trees.

Neural networks - These methods create non-linear compositions of functions that allow to express a variety of relationships. Often called *deep neural networks*, these methods have gained massive popularity in recent years.

In the four examples below, Listing 9.4, Listing 9.5, Listing 9.6 and Listing 9.7 we obtain the MNIST dataset via the Flux package. We then train the classifier based on the 60,000 training images and test it on the 10,000 testing images.

Linear Least Squares Classification

One of the most simple classifiers that we can create is based on least squares. We consider each image as a vector and obtain different least squares estimators for each type of digit. For digit $\ell \in \{0, 1, 2, \dots, 9\}$ we collect all the training data vectors, with $y_i = \ell$. Then for each such i , we set $y_i = +1$ and for all other i with $y_i \neq \ell$, we set $y_i = -1$. This labels our data as classifying “yes digit ℓ ” vs. “not digit ℓ ”. Call this vector of -1 and $+1$ values $y^{(\ell)}$ for every digit ℓ . We then compute,

$$\beta^{(\ell)} = A^\dagger y^{(\ell)} \quad \text{for } \ell = 0, 1, 2, \dots, 9, \quad (9.7)$$

where A^\dagger is the pseudo-inverse associated with the 60,000 images. It is the pseudo-inverse of the $60,000 \times 785$ matrix A (allowing also a first column of 1’s for a *bias term*). Now for every image i , the inner product $\beta^{(\ell)} \cdot x_i$ yields an estimate of how likely this image is of the digit ℓ . A very high value indicates a high likelihood and a low value is a low likelihood. We then classify an arbitrary image \tilde{x} by selecting,

$$\hat{y}(\tilde{x}) = \arg \max \beta^{(\ell)} \cdot \tilde{x}. \quad (9.8)$$

Observe that during training, this classifier only requires calculating the pseudo-inverse of A once. It then only needs to remember 10 vectors of length 785, β^0, \dots, β^9 . Then based on these 10 vectors, a decision rule is very simple to execute in (9.8).

We illustrate this in Listing 9.4 where we achieve 86% accuracy. We also output the *confusion matrix* in the output. This matrix shows for each real label (row), how many labels were classified (column). It is a count over the 10,000 test images.

Listing 9.4: Linear least squares classification

```

1  using Flux.Data.MNIST, PyPlot, LinearAlgebra
2  using Flux: onehotbatch
3
4  imgs    = MNIST.images()
5  labels  = MNIST.labels()
6  nTrain  = length(imgs)
7

```

```

8  trainData = vcat([hcat(float.(imgs[i])...) for i in 1:nTrain]...);
9  trainLabels = labels[1:nTrain];
10
11 testImgs = MNIST.images(:test)
12 testLabels = MNIST.labels(:test)
13 nTest = length(testImgs)
14
15 testData = vcat([hcat(float.(testImgs[i])...) for i in 1:nTest]...);
16
17 A = [ones(nTrain) trainData];
18 Adag = pinv(A);
19 tfPM(x) = x ? +1 : -1
20 yDat(k) = tfPM.(onehotbatch(trainLabels,0:9)'[:,k+1])
21 bets = [Adag*yDat(k) for k in 0:9];
22
23 classify(input) = findmax([(1 ; input)']*bets[k] for k in 1:10])[2]-1
24
25 predictions = [classify(testData[k,:]) for k in 1:nTest]
26 confusionMatrix = [sum((predictions .== i) .& (testLabels .== j))
27                      for i in 0:9, j in 0:9]
28 accuracy = sum(diag(confusionMatrix))/nTest
29
30 println("Accuracy: ", accuracy)
31 println("Confusion Matrix:")
32 confusionMatrix

```

Accuracy: 0.8603

Confusion Matrix:

```

10x10 Array{Int64,2}:
 944      0     18      4      0     23     18      5     14     15
    0   1107    54    17    22    18    10    40    46    11
    1     2   813    23      6      3      9     16     11      2
    2     2    26   880      1    72      0      6    30    17
    2     3    15      5   881     24     22     26    27    80
    7     1     0    17      5   659     17      0    40      1
   14     5    42      9    10    23   875      1    15      1
    2     1    22    21      2    14      0   884     12    77
    7    14    37    22     11    39      7      0   759      4
    1     0     5    12     44     17      0    50     20   801

```

- In lines 4-6 we load the training images, the labels and determine nTrain as the number of training images.
- Line 8 converts the training images into a big matrix `trainData`.
- Lines 11-15 deal with the test images in a similar manner.
- In line 17 we construct the matrix A . This is followed by line 18 where we compute A^\dagger .
- In line 19 we construct a simple function, `tfPM()`, that converts true or false values to $+1$, -1 respectively. In line 20 we use the `onehotbatch()` function from package Flux. It converts each of the training labels into an array of length 10 comprised of true/false values where only a single entry of the array is true, matching the location of the digit. This then creates `yDat(k)` once `tfPM()` is applied. It is $y^{(\ell)}$ as described above.

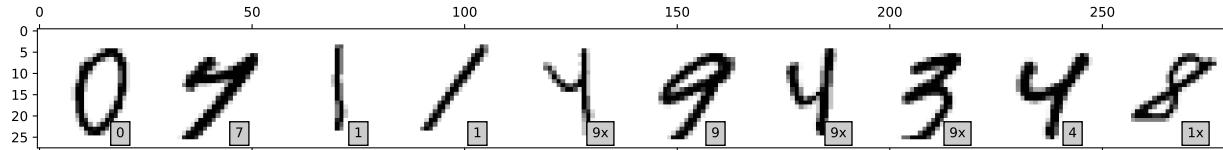


Figure 9.3: Sample images from the test set for SVM. Classified label presented on the bottom right of each digit image with “x” if error.

- Line 21 executes the estimation implementing (9.7).
- Our classifier is then implemented in line 23 according to (9.8).
- In line 25 we use the classifier to create predictions for the test data. We then compute confusionMatrix in line 26 and accuracy in line 27.

Support Vector Machines

An alternative classifier can be constructed using *Support Vector Machines (SVM)*. We omit the details of how this classifier works and refer the reader to the literature. Instead we use the LIBSVM package and carry out a classification experiment on MNIST. Listing 9.5 carries out the classification and prints out the accuracy. It also creates Figure 9.3, for the purpose of illustrate some digits that are classified correctly and some that are not.

Listing 9.5: Support vector machines

```

1  using Flux.Data.MNIST, LIBSVM, PyPlot
2
3  imgs     = MNIST.images()
4  labels   = MNIST.labels()
5
6  trainData = hcat([vcat(float.(imgs[i])...) for i in 1:1000]...)
7  trainLabels = labels[1:1000];
8
9  testData = hcat([vcat(float.(imgs[i])...) for i in 1001:2000]...)
10 testLabels = labels[1001:2000];
11
12 model = svmtrain(trainData,trainLabels);
13
14 (predicted_labels, decision_values) = svmpredict(model, testData);
15
16 accuracy = sum(predicted_labels .== testLabels)/1000
17 println("Prediction accuracy (measured on test set of size 1000): ", accuracy)
18
19 showImages = float.(imgs[1001:1010])
20 matshow(hcat(showImages...), cmap="Greys")
21 for i in 1:10
22     ok = predicted_labels[i] == testLabels[i] ? "" : "x"
23     annotate("$ (predicted_labels[i]) $(ok)", xy=(28i-10, 25), xytext=(28i-10, 25),
24             bbox=Dict("fc"=>"0.8"))
25 end
```

```
Prediction accuracy (measured on test set of size 1000): 0.829
```

- Lines 6-10 construct the test data, train data and the respective labels.
- Training of the model is carried out in line 12 via `svmtrain()`.
- Prediction is carried out in line 14 via `svmpredict()`.
- The accuracy is computed and printed out in lines 16-17.
- Lines 19-25 create Figure 9.3.

Decision Trees and Random Forest

An alternative general purpose classifier is the *random forest* algorithm. It is a *bagging algorithm* applied to *decision trees*. We omit the details and refer the reader to the literature. Instead we just use the `DecisionTree` Julia package. This is carried out in Listing 9.6 below.

Listing 9.6: Random forest

```

1  using Flux.Data.MNIST, DecisionTree, PyPlot, Random
2  Random.seed!(1)
3
4  imgs    = MNIST.images()
5  labels = MNIST.labels()
6
7  trainData = vcat([hcat(float.(imgs[i])...) for i in 1:50000]...)
8  trainLabels = labels[1:50000];
9
10 testData = vcat([hcat(float.(imgs[i])...) for i in 50001:60000]...)
11 testLabels = labels[50001:60000];
12
13 model = build_forest(trainLabels, trainData, 10, 40, 0.7, 10)
14
15 predicted_labels = [apply_forest(model, testData[k,:]) for k in 1:10000]
16
17 accuracy = sum(predicted_labels .== testLabels)/10000
18 println("Prediction accuracy (measured on test set of size 100): ", accuracy)
19
20 k = 1
21 while predicted_labels[k] == testLabels[k]
22     global k +=1
23 end
24 println("Example error (MNIST image $(50000+k)):",
25         " Predicted $(predicted_labels[k]) but it is actually $(testLabels[k]).")
```

```
Prediction accuracy (measured on test set of size 100): 0.9379
Example error (MNIST image 50006): Predicted 9 but it is actually 4.
```

- Lines 7-11 setup the train data and the test data similar to previous examples.

- In line 13 we use the `build_forest()` function to run a random forest algorithm on the `trainData` with `trainLabels`. The values 10, 40, 0.7 and 10 passed as further arguments are hyper-parameters.
- In line 15 we use the `apply_forest()` with the model created on `testData`. This creates the `predicted_labels` array.
- The remainder of the code prints the accuracy and also seeks a test label where there was a prediction error and prints it.

Deep Neural Networks

Arguably, one of the most popular methods of machine learning in recent years is the *neural network* or *deep neural networks*. Although it embodies several different concepts, it fundamentally refers to a combination of matrices/tensors (often called *neurons*) and the operations between them. Data (i.e. predictor variable values/features) are input to the neural network, which then parse these values, and output predictions based on them. In the case of supervised learning, these predictions are compared to the response variable, and checked for their accuracy. Since the values/weights of the matrices/neurons are initialized randomly, the first parse of the data will have a low level of accuracy - in fact one would expect the accuracy of the first predictions to be no better than random chance. However, by making small random change to the values/weights of the neurons, keeping the changes which increase the accuracy of the predictions, and iterating this process several thousand times, the neural network can be *trained* so that it can accurately predict future responses based on unseen data. The specific process of training a neural network involves the minimization of a *loss function*. In fact, *stochastic gradient descent* (SGD) introduced in Section 8.1 is often used.

Since training a neural network takes many iterations, and the majority of operations are based on linear algebra, it is common to train neural networks through the use of graphics processing units (GPU's). Some companies have even developed application specific hardware, such as Google's Tensor Processing Units (TPU's), specifically designed to run machine learning algorithms. There are many different machine learning language frameworks available, including Google's *TensorFlow*, Caffe, Torch, Pytorch, and others. There are also high level libraries which aim to simplify the often repetitive nature of setting up the different neural networks, such as Keras which acts as a wrapper for Google Tensorflow.

We now explore a neural network. When it comes to neural networks in Julia, the `FLUX.jl` package comes as an aid. Written entirely in Julia, it offers both high-level, and low-level functionality, along with the ability to be run on either a CPU or GPU.

The *neural networks architecture*, is the specific arrangement of the matrices/layers of the network, the operations performed between them, and various other parameters of the network. In addition to the number of layers and their inter-linked arrangement chosen, there are several other high level parameters, called *hyperparameters*, which often have to be 'tuned' in order for a neural network produce accurate predictions. These parameters must also be decided by the programmer, and include,

- The *activation function* - this function determines how the weights/values of the layers are updated at each iteration. Typically each layer of the network has its own activation function.

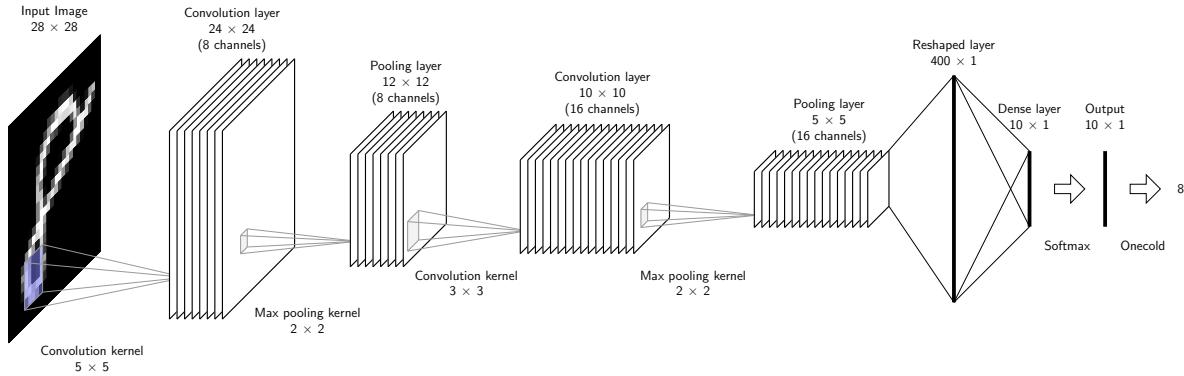


Figure 9.4: The architecture of the neural network in Listing 9.7.

Several different activation functions exist, including sigmoid, tanh, relu (one of the more popular choices), leaky relu, and others.

- The *objective function* - also sometimes called the *loss function* or *error function*, is used to quantify the difference/error between the neural networks predictions, and the observed values. As the values/weights of the network are changed, so does the loss function, and through the minimization of this function, the neural network ‘learns’ a series of internal weights that result in good predictive results.
- The *optimization function* - the optimization function is a type of algorithm which is used to determine how the weights within the network should be updated. It essentially calculates the partial derivative of the loss function with respect to the internal weights, which are then updated in the opposite direction (with the objective of decreasing the loss function above). One type of optimizer algorithm is SGD (already introduced). Another common one used is the *ADAM optimizer*, which is in fact an extension of SGD.

We now explore these concepts through a practical neural network example. In this example, we create a simple neural network for the purpose of the classification of images of hand drawn digits 0 to 9. The example provided here is a modified version of the MNIST example from the FLUX.jl model zoo, and is based on supervised learning of the *MNIST* data set, which is a dataset consisting of 60,000 images of hand drawn digits 0 to 9. Each image is 28×28 pixels in size, and has a corresponding label, or record, of what digit the writer intended to write. The first hand drawn image, labelled as a ‘five’ is shown in 9.1. See also [Inn18].

The choice of architecture we use for our neural network example is shown in Figure 9.4. It consists primarily of four layers with multiple channels; a convolutional layer, a softmax pooling layer, another convolutional layer, and another softmax pooling layer. This last layer is then reshaped into a 1-dimensional array, then fully convoluted and parsed to a smaller array (this *fully convoluted layer* is also called a *dense layer*). Finally, the *softmax function* is used on this layer/array, and the *one cold encoding* function is used to select the digit (one of 0 to 9) with the highest value as the predicted output. In Listing 9.7 the neural network is constructed and trained, based on the approach detailed previously.

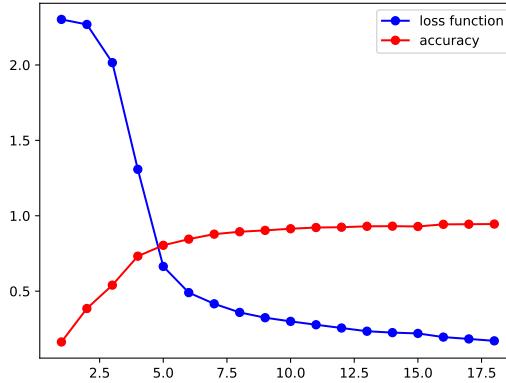


Figure 9.5: Loss function and accuracy of the neural network.

Listing 9.7: A convolutional neural network

```

1  using Flux, Flux.Data.MNIST, Random, Statistics, PyPlot
2  using Flux: onehotbatch, onecold, crossentropy, throttle, @epochs
3  using Base.Iterators: repeated, partition
4  Random.seed!(1)
5
6  imgs    = MNIST.images()
7  labels = onehotbatch(MNIST.labels(), 0:9)
8
9  train   = [(cat(float.(imgs[i])..., dims = 4), labels[:,i])
10           for i in partition(1:50000, 1000)]
11  test    = [(cat(float.(imgs[i])..., dims = 4), labels[:,i])
12           for i in partition(50001:60000, 1000)]
13
14 m = Chain(
15     Conv((5,5), 1=>8, relu),
16     x -> maxpool(x, (2,2)),
17     Conv((3,3), 8=>16, relu),
18     x -> maxpool(x, (2,2)),
19     x -> reshape(x, :, size(x, 4)),
20     Dense(400, 10), softmax)
21
22 loss(x, y) = crossentropy(m(x), y)
23 accuracy(x, y) = mean(onecold(m(x)) .== onecold(y))
24 opt = ADAM(params(m))
25
26 L, A = [], []
27 evalcb = throttle(10) do
28     push!( L, mean( [ loss( test[i][1], test[i][2] ).data for i in 1:10] ) )
29     push!( A, mean( [ accuracy( test[i][1], test[i][2] ) for i in 1:10] ) )
30 end
31
32 @epochs 3 Flux.train!(loss, train, opt, cb=evalcb)
33
34 plot( 1:length(L) , L, label="loss function", "bo-")
35 plot( 1:length(A) , A, label="accuracy", "ro-");

```

- In lines 1 to 3 the various packages and functions required for this example are called. We

call the main Flux package, and the MNIST dataset from it. We also call the `onehotbatch`, `onecoldbatch` and `crossentropy` functions, along with the `throttle` function, and the `@epoch` macro. We explain the use of these functions later in this example as they are used.

- In line 4 we seed the random number generator.
- In line 6 the image data of the MNIST dataset is stored as `imgs`. There are a total of 60,000 images, each of which is a 28×28 array of values.
- In line 7 the labels of the MNIST dataset are first one-hot encoded via the `onehotbatch` function. This function takes each label (a number 0 to 9) and converts it to a 10-dimensional true/false array, with `true` recorded at the index of the label. The resulting $10 \times 60,000$ matrix is stored as `labels`, and each column corresponds to the classification of each image in `imgs`.
- In lines 9 to 12 we divide the images and corresponding labels into two datasets. The first 50,000 observations are assigned to `train`, while the last 10,000 are assigned to `test` `test` via the use of comprehensions. As part of this process, the `partition` function is used to batch into groups of 1000 observations each. For each batch, the data for the images are paired together with their corresponding labels as tuples. The data for each pixel of each image is converted to a `float` value between 0 and 1, based on the handwriting of the image.
- The `cat()` function is used along with the splat operator (`....`) to convert each 28×28 image into a four-dimensional array. This is done as the neural network accepts input data in format WHCN - which stands for width, height, number of channels, and size of each batch. Hence here the image data for each batch is shaped into a $28 \times 28 \times 1 \times 1000$ array (since we specified a batch size of 1000). Note that, as mentioned above, for each batch this data is then grouped as a tuple, with the images corresponding labels.
- As a side point, note that `i` for each batch is actually an array, and this approach is used in order to preserve the 28×28 shape of the data in the first two dimensions (i.e. `cat(float.(imgs[[1]]), ..., dims = 4)` returns $28x28x1x1$).
- In lines 14 to 20 we define the architecture of the neural network. The `Chain()` function is used to chain multiple layers together, by performing operations sequentially in order from left to right. The resulting architecture stored as the variable `m`, and Figure 9.4 can be referred to for a visual representation.
- In line 15 the first convolutional layer is created via the `Conv()` structure. The first argument is a tuple, representing the dimensions of the kernel to be passed over the input image. The second argument `1=>8` maps the one channel of input to 8 channels of output. The third argument is the activation function, here we specify rectified linear unit via `relu`. Note `Conv()` also takes `stride` as an optional argument, and uses 1 as default. In addition, no padding was specified so defaults to false. Hence by parsing our 5×5 image kernel over our 28×28 image, we obtain 8 channels of size 24×24 ($28 - (5 - 1) = 24$).
- In line 16 the anonymous function `x -> maxpool(x, (2, 2))` is used for the second layer. This `maxpool` function parses a 2×2 kernel over each of the 8 channels, and at each point pools the values into a single value. Hence the resulting output is 8 channels of size 12×12 .

- In line 17 a second convolution kernel is used in a similar approach to that of line 15. This time a 3x3 kernel is used, and the number of input and output channels specified as 8 and 16 respectively. Hence the output after this operation is 16 channels of size 10×10 ($12 - (3 - 1)$).
- In line 18 a second maxpool function is parsed over our 16 channels. A 2x2 kernel is used, hence resulting in 16 5x5 channels.
- In line 19 the 16 channels are flattened into a single array via the `reshape()` function. Since there are 16 5x5 channels, the resulting object is a single 400×1 channel/array. Note that `size(x, 4)` is the second dimension of the reshape (it is in fact the batch number, 1000).
- In line 22 the `Dense()` function is used to create a fully connected layer, effectively mapping the previous 400×1 array to a 10×1 array (the two arguments are the input output dimensions respectively). Finally, the `softmax()` function is applied to the dense layer. This function takes the log-probabilities of the values in the dense layer and outputs 10×1 array of probabilities which sum to 1.
- In line 22 the `loss` function is defined. It takes two arguments, a batch of features, `x`, and corresponding labels, `y`. The function parses the features through the neural network `m()`, and calculates the crossentropy between the neural networks predictions and the labels of the data.
- In line 23 the `accuracy()` function is created. It is somewhat similar to the `loss` function of line 22, however uses the `onecold()` function (the inverse of one hot encoding) to compare the models predictions directly with the data labels, and then calculates the resulting mean.
- In line 24 the ADAM optimizer is specified. This optimizer takes a parameter list, which includes *learning rate* and *weight decay*, among others. In our case we use the `params()` function is used to get the parameters of the model `m`, and these parameters are used as the argument of the optimizer. The optimizer is stored as `opt`.
- In line 32 the model is trained by calling `Flux.train!`. This function takes a minimum of three arguments, an objective function, some training data, and an optimizer. Here we use the `loss` function previously defined, the training data `train`, and the optimizer defined previously `opt`. By default, the function will loop over all of the batches once, and update the internal weights after each batch (a single pass over all of a training dataset is called an *epoch*). Due to the small size of many datasets, it is common to loop over the same data multiple times in order to further train the model and increase its accuracy. Therefore the `@epochs` macro is used to train the data over multiple epochs, in this case 3.
- Finally, a fourth argument is specified in `Flux.train!`, specifically a callback, in which the value of the loss function, and accuracy of the network at various steps throughout training is saved.
- These values of the loss function and accuracy of the network are displayed in Figure 9.5. It can be seen that at the beginning of training, the accuracy of the model is around 10%, which is the accuracy we would expect given . However, as the batches are parsed by the model, and the internal weights of the layers are updated, we can see that the loss function decreases. Indeed it is the minimization of this loss function that drives the updating of these weights. As the model is ‘trained’, we also observe that the accuracy increases, and plateaus out.

Note that the accuracy here is calculated on the unseen test data `test`, i.e. only the training data is used to train the network - the test data is purely for checking model accuracy. As we can see, as the loss function decreases, the accuracy of the model increases, until they both plateau out. Hence we arrive at a neural network that has a good level of prediction of unseen data.

9.5 Unsupervised Learning Methods

Data is not always labelled, i.e. the features of a dataset are not always classified, or the labels themselves may be unknown. In these cases of *unlabelled data*, the goal is to identify patterns in the underlying features, such that new observations with similar features can be grouped accordingly, and some overall conclusions drawn. This is known as *unsupervised learning*, and common examples include various forms of *clustering* or *dimension reduction*.

For a dataset X_1, \dots, X_n , clustering is the act of associating a cluster ℓ with each observation, where ℓ comes from a small finite set. That is, clustering considers the data and outputs a function $c(x)$ with datapoints in the domain and a range of $\{1, \dots, k\}$. The ℓ 'th cluster is then,

$$C_\ell = \{X_i \text{ with } i \in \{1, \dots, n\} \mid c(X_i) = \ell\}.$$

A clustering algorithm attempts to choose the clusters such that the elements of each C_ℓ are as homogenous as possible.

A dimension reduction algorithm attempts to create a transformed dataset $\tilde{X}_1, \dots, \tilde{X}_n$ where each \tilde{X}_i is of lower dimension than X_i , yet the information embodied in the new dataset is similar to the information in the original dataset. Good dimension reduction is able to significantly reduce the size of each X_i while at the same time maintaining the main attributes of the dataset.

For clustering we consider two types of algorithms: *k-means* and *hierarchical clustering*. For dimension reduction we consider *principal component analysis (PCA)*.

***k*-Means Clustering**

When using *k*-means clustering, we assume that the data points, X_1, \dots, X_n are each vectors in p -dimensional Euclidean space. We then specify a number k , determining the number of clusters that we wish to find. We then seek the function $c(x)$ (or a partition C_1, \dots, C_k) together with means of clusters, J_1, \dots, J_k with an aim of minimizing,

$$\sum_{\ell=1}^k \sum_{x \in C_\ell} \|x - J_\ell\|^2. \quad (9.9)$$

Such a minimization is generally computationally challenging, however it can be approximately achieved by separating the problem into two components.

Mean computation: Given $c(x)$, finding the means J_1, \dots, J_ℓ is simply done by setting,

$$J_\ell = \frac{1}{|C_\ell|} \sum_{x \in C_\ell} x. \quad (9.10)$$

This is the element-wise average (over the p elements) over all the vectors in C_ℓ .

Labelling: Given, J_1, \dots, J_ℓ finding $c(x)$ that minimizes (9.9) is done by setting,

$$c(x) = \arg \min_{\ell} \|x - J_\ell\|, \quad (9.11)$$

i.e. the label of each element is determined by the closest mean in Euclidean space.

The k -means algorithm operates by iterating over the mean computation step, (9.10), followed by the labelling step (9.11). This is done until no more changes are made to the labels and the means. Such an iteration generally doesn't minimize the objective (9.9), however this approximation is often satisfactory.

As discussed above, datasets can sometimes be unlabelled, and consist simply of a set of features. In such cases, one approach is to attempt to cluster the observations into one of k groups based on these features. Here the *k -means clustering* method comes as an aid.

We now consider the `xclara` dataset from the `clusters` RDataSets package. This dataset comprises observations consisting of two variables `V1` and `V2`. We set $k = 3$ and carry out k -means in two separate code examples. In Listing 9.8 we use the `Clustering` package. This listing also generates Figure 9.6. We then implement k -means from scratch for this example in Listing 9.9. Since we start with the same initial conditions in both code examples, both examples yield the same clustering (up to ordering of the labels). This is evident via the number of observations found in each cluster.

Listing 9.8: Carrying out k -means via the Clustering package

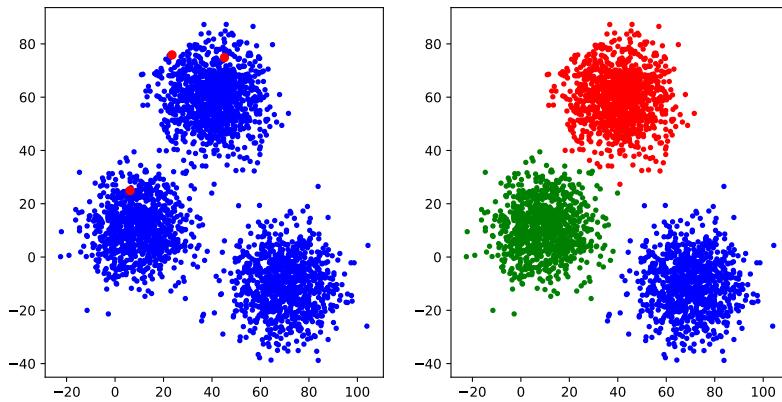
```

1  using Clustering, RDatasets, PyPlot
2
3  df = dataset("cluster", "xclara")
4  data = copy(convert(Array{Float64}, df)')
5
6  seeds = initseeds(:rand, data, 3)
7  xclara_kmeans = kmeans(data, 3)
8
9  println("Number of clusters: ", nclusters(xclara_kmeans))
10 println("Counts of clusters: ", counts(xclara_kmeans))
11
12 df[:Group] = assignments(xclara_kmeans)
13
14 fig = figure(figsize=(10, 5))
15 subplot(121)
16 plot(df[:, :V1], df[:, :V2], "b.")
17 plot(df[seeds, :V1], df[seeds, :V2], markersize=12, "r.",)
18
19 subplot(122)
20 plot( df[df.Group == 1, :V1], df[df.Group == 1, :V2], "b." )
21 plot( df[df.Group == 2, :V1], df[df.Group == 2, :V2], "r." )
22 plot( df[df.Group == 3, :V1], df[df.Group == 3, :V2], "g." )
```

```

Number of clusters: 3
Counts of clusters: [952, 1149, 899]
```

- In line 3 the dataset `xclara` from the `clusters` package from `RDataSets` is stored as `df`.

Figure 9.6: Plot of k -means clustering, for $k = 3$.

- In line 4 the data frame `df` is converted to an array of `Float64` type in order to remove the missing type of the data frame for compatibility. The array is then transposed, as it needs to be in this format for the `kmeans` function which is used on line 6.
- In line 6 the `initseeds()` function is used to randomly select the three original centroids to be used in the k -means calculation. This function takes three arguments, the method of selection of the starting centroids :`rand`, the dataset to be performed on `data`, and the number of clusters specified (in this case 3).
- In line 7 the `kmeans()` function from the `Clustering` package is used to perform k -means clustering on the dataset.
- In line 9 the `nclusters()` function is used to return the number of clusters of the k -means output.
- In line 10 the `counts()` function is used to return the number of observations in each of the clusters of the k -means output.
- The remaining lines are used to create Figure 9.6. Note that in line 16 the original seed centroids are plotted in the first subplot.

Listing 9.9: Manual implementation of k -means

```

1  using RDatasets, PyPlot, Distributions, Random
2  Random.seed!(1)
3
4  k = 3
5
6  xclara = dataset("cluster", "xclara")
7  n,_ = size(xclara)
8  dataPoints = [convert(Array{Float64,1},xclara[i,:]) for i in 1:n]
9  shuffle!(dataPoints)
10
11 xMin,xMax = minimum(first.(dataPoints)),maximum(first.(dataPoints))
12 yMin,yMax = minimum(last.(dataPoints)),maximum(last.(dataPoints))
13

```

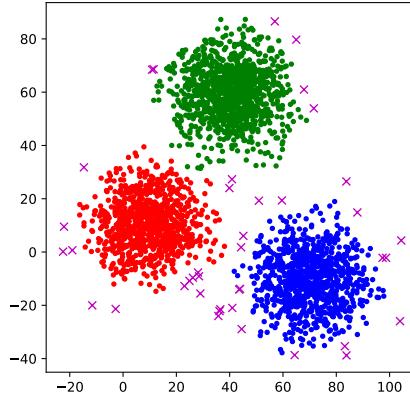


Figure 9.7: Three main clusters arising after 30 iterations of hierarchical clustering with several other small clusters (data points marked by 'X').

```

14 means = [[rand(Uniform(xMin,xMax)),rand(Uniform(yMin,yMax))] for _ in 1:k]
15 labels = rand([1,k],n)
16 prevMeans = -means
17
18 while norm(prevMeans - means) > 0.001
19     prevMeans = means
20     labels = [findmin([norm(means[i]-x) for i in 1:k])[2] for x in dataPoints]
21     means = [sum(dataPoints[labels == i])/sum(labels ==i) for i in 1:k]
22 end
23
24 cnts = [sum(labels == i) for i in 1:k]
25
26 println("Counts of clusters (manual implementation): ", cnts)

```

Counts of clusters (manual implementation): [899, 952, 1149]

- In lines 11 and 12 we find a bounding box for the data points for selecting random starting means.
- In lines 14 and 15 we randomly initialize the $c(x)$ and J_1, \dots, J_k .
- The main iteration is in lines 18-22 where in line 20 we implement (9.11) and in line 21 we implement (9.10).

Hierarchical Clustering

An alternative form of clustering is *hierarchical clustering*. This type of algorithms comes in several variants, including *agglomerative*, *divisive* and several variants for measuring the distance between observations. In Listing 9.10 we use the `hclust()` function from package `Clustering` to carry out divisive hierarchical clustering on the `xclara` dataset.

Listing 9.10: Carrying out hierarchical clustering

```

1  using RDatasets, PyPlot, Clustering, Random, LinearAlgebra
2  Random.seed!(1)
3
4  xclara = dataset("cluster", "xclara")
5  n,_ = size(xclara)
6  dataPoints = [convert(Array{Float64,1},xclara[i,:]) for i in 1:n]
7  shuffle!(dataPoints)
8  D = [norm(pt1 - pt2) for pt1 in dataPoints, pt2 in dataPoints];
9
10 result = hclust(D)
11 for K in 1:30
12     clusters = cutree(result,k=K)
13     println("K=$K: ", [sum(clusters .== i) for i in 1:K])
14 end
15
16 cluster(ell,K) = (1:n)[cutree(result,k=K) .== ell]
17
18 C1,C2,C3 = cluster(1,30),cluster(2,30),cluster(3,30)
19 fig = figure(figsize=(5, 5))
20 plot(first.(dataPoints[C1]),last.(dataPoints[C1]),"b.")
21 plot(first.(dataPoints[C2]),last.(dataPoints[C2]),"r.")
22 plot(first.(dataPoints[C3]),last.(dataPoints[C3]),"g.")
23 for ell in 4:30
24     clst = cluster(ell,30)
25     plot(first.(dataPoints[clst]),last.(dataPoints[clst]),"mx")
26 end

```

```

1:[3000]
2:[2999, 1]
3:[2997, 2, 1]
4:[2997, 1, 1, 1]
5:[2996, 1, 1, 1, 1]
6:[2995, 1, 1, 1, 1, 1]
7:[2994, 1, 1, 1, 1, 1, 1]
8:[2993, 1, 1, 1, 1, 1, 1, 1]
9:[2989, 4, 1, 1, 1, 1, 1, 1, 1]
10:[2988, 4, 1, 1, 1, 1, 1, 1, 1, 1]
11:[2986, 2, 4, 1, 1, 1, 1, 1, 1, 1, 1]
12:[2985, 2, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1]
13:[2984, 2, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
14:[2983, 2, 1, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
15:[2982, 2, 1, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
16:[2981, 2, 1, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
17:[2980, 2, 1, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
18:[941, 2039, 2, 1, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
19:[939, 2039, 2, 1, 4, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
20:[937, 2039, 2, 2, 1, 4, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
21:[937, 2038, 2, 2, 1, 4, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
22:[937, 2037, 1, 2, 2, 1, 4, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
23:[937, 2032, 1, 2, 2, 1, 4, 1, 5, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
24:[935, 2032, 1, 2, 2, 1, 4, 2, 1, 5, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
25:[935, 2030, 1, 2, 2, 1, 4, 2, 1, 5, 2, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1]
26:[933, 2030, 1, 2, 2, 1, 4, 2, 1, 5, 2, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1]
27:[933, 2028, 1, 2, 2, 1, 4, 2, 1, 5, 2, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 1, 1, 1]
28:[933, 882, 1146, 1, 2, 2, 1, 4, 2, 1, 5, 2, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 1, 1, 1]
29:[932, 882, 1146, 1, 2, 2, 1, 4, 2, 1, 5, 2, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 1, 1, 1]
30:[932, 882, 1146, 1, 2, 2, 1, 3, 2, 1, 5, 2, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 1, 1, 1]

```

- In lines 4-8 we read the dataset and create the matrix D that records the Euclidean distances between each data point.
- Line 10 is where the `hclust()` function from the `Clustering` package is called. It runs a hierarchical clustering algorithm on the distance matrix D .
- In lines 11-14 we print the hierarchical clustering tree. In each iteration of line 12, the `cutree()` function is used to return K clusters in an array, `clusters` that contains the cluster associated with each data point. In line 13, we print the number of elements in each cluster. As is observed from the output, up to the 28'th iteration, there are only two main clusters. Only in the 28'th iteration the algorithm discovers 3 main clusters.
- Lines 18-26 plot Figure 9.7. The function `cluster()` defined in line 16, returns all of the indexes for cluster `e11`. We use it for plotting in the code that follows.

Principal Component Analysis

Say we are presented with a dataset of vector observations X_1, \dots, X_n , each of dimension p , where $n > p$. The archetypical algorithm for dimensionality reduction is *Principal Component Analysis (PCA)*. The idea is to choose a suitable dimension $k < p$ and create k dimensional vectors $\tilde{X}_1, \dots, \tilde{X}_n$ where for every observation i ,

$$\tilde{X}_i = V X_i, \quad (9.12)$$

with V a $k \times p$ matrix. Each row of V is called a principal component as it takes a linear combination of X_i and creates a coordinate in \tilde{X}_i . The act of carrying out PCA is the act of determining k , finding V and analyzing the reduced dataset $\tilde{X}_1, \dots, \tilde{X}_n$.

One way to consider PCA is as an advanced *data visualization* technique. For example, consider the MNIST dataset where $p = 784$. We can view one image, X_i by plotting the image as in Figure 9.1, however how can we view thousands of images together? For this, if we reduce the dimension of an image from 784 to $k = 2$ we are able to create plots like Figure 9.9. In such a plot, the full information of every image is clearly not present, still we may see how the image compares to others. Here the matrix V of (9.12) is 2×784 dimensional and it consists of two principal components. It is computed from the full data consisting of 60,000 images, each with 784 pixels.

There are different ways to compute V and determine k . One way is to consider the sample covariance matrix, $\hat{\Sigma}$, associated with the data as presented in Listing 4.8 of Chapter 4. Since it is a symmetric matrix, all eigenvalues are real and there are corresponding orthonormal eigenvectors. Hence we may *diagonalize* it via $\hat{\Sigma} = M \text{diag}(\lambda_1, \dots, \lambda_p) M'$, where M is a vector of column eigenvectors, v_1, \dots, v_p , with corresponding eigenvalues $\lambda_1, \dots, \lambda_p$. Assume also that $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_p$. We may then represent $\hat{\Sigma}$ via the sum of *rank one matrices* and for $k < p$

$$\hat{\Sigma} = \sum_{i=1}^p \lambda_i v_i v_i' \approx \sum_{i=1}^k \lambda_i v_i v_i'.$$

We then set the matrix V to consist of the rows v_1', \dots, v_k' . By ordering the eigenvalues, we are able to choose the significant ones first and make a judgement call on k . This is sometimes done with an aid of a *scree plot* that plots the diminishing contribution of each λ_i . Listing 9.11 uses the `MultivariateStats` package to carry out PCA.

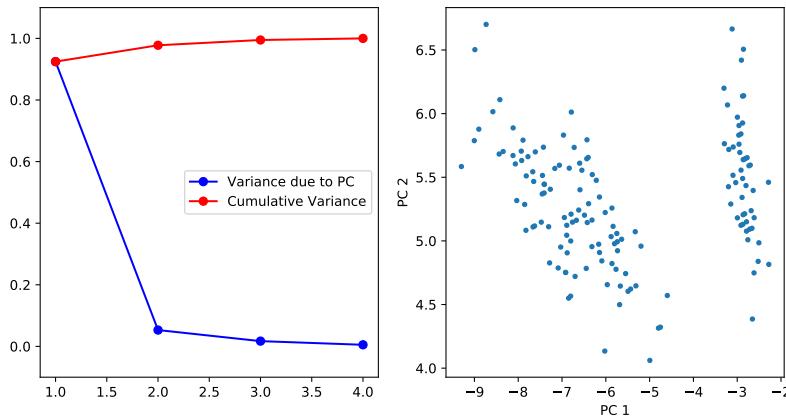


Figure 9.8: A scree plot, along with the result of principal component analysis.

Listing 9.11: Principal component analysis

```

1  using Statistics, MultivariateStats, RDatasets, PyPlot, LinearAlgebra
2
3  data = dataset("datasets", "iris")
4  data = data[:SepalLength,:SepalWidth,:PetalLength,:PetalWidth]
5  n = size(data)[1]
6  x = convert(Array{Float64,2},data)'
7
8  model = fit(PCA, x, maxoutdim=4, pratio = 0.999)
9  M = projection(model)
10
11 function manualProjection(x)
12     covMat = cov(x')
13     ev = eigvals(covMat)
14     eigOrder = sortperm(eigvals(covMat), rev=true)
15     eigvecs(covMat)[:,eigOrder]
16 end
17
18 println("Manual vs. package: ", maximum(abs.(M-manualProjection(x))))
19
20 pcVar = principalvars(model) ./ tvar(model)
21 cumVar = cumsum(pcVar)
22
23 pcDat = M[:,1:2]' *x
24
25 figure(figsize=(10,5))
26 subplot(121)
27 plot( 1:length(pcVar) , pcVar,"bo-", label="Variance due to PC")
28 plot( 1:length(cumVar) , cumVar,"ro-", label="Cumulative Variance");
29 legend(loc="center right")
30 ylim(-0.1,1.1)
31
32 subplot(122)
33 plot(pcDat[1,:],pcDat[2,:],".")
34 xlabel("PC 1"); ylabel("PC 2");

```

- In lines 3-6 we read the `iris` dataset and consider the four numerical variables in the dataset.

We then create a 4×150 dimensional matrix of the data, \mathbf{x} where each column is a 4 dimensional data point.

- Line 8 uses the `fit()` function on PCA, as defined in the `MultivariateStats` package. The `maxoutdim` setting is redundant in our case, however in other cases can be used to limit the number of principal components obtained. The `pratio` setting indicates to stop when the cumulative variance is greater than 0.999. The default is 0.99. The resulting `model` object can then be queried.
- Line 9 uses the `projection()` function from the `MultivariateStats` package. It returns a matrix where each column is a principal component.
- We define our own function `manualProjection()` in lines 11-16. This function creates a matrix with columns as principal components, analogously to the matrix `M` created in line 9. We compute the sample covariance matrix in line 12 and then compute its eigenvalues in line 13. Then our use of `sortperm()` with `rev=true` returns the permutation of eigenvalues from highest to smallest (the covariance matrix is guaranteed to be symmetric and hence the eigenvalues are real). We then compute corresponding eigenvectors with `eigvecs()` in line 15. These are in a matrix with each column an eigenvector. We then reshuffle the columns according to the `eigOrder` previously computed. This is the matrix of principal components.
- The matrix of principal components resulting from `manualProjection()` is the same as the matrix that was generated in line 9. We illustrate this in line 18 where we print the maximum absolute difference of entries.
- In line 20 we use the `principalvars()` function on `model`. It returns the variances associated with each principal component. This can also be obtained within `manualProjection()` if we wished by evaluating `ev[eigOrder]`. These values are then normalized by dividing by the scalar `tvar(model)`. We accumulate these values in line 21 when we use the builtin `cumsum()` function. The `pcVar` array and the associated `cumVar` array are then plotted in lines 26-30, creating a *scree plot*.
- In line 23 we decide to use 2 principal components and hence select the first two columns of `M` via `1:2`. Applying the transpose of this matrix to the data `x` yields a 2×150 matrix where each column is a principal component. These are then plotted in the second figure in lines 32-34.

PCA is often used in conjunction with other algorithms, including supervised learning algorithms. We may often pre-process the data and train an algorithm to classify based on principal components instead of the original data.

We don't illustrate interaction of PCA and supervised learning here, however we hint at the power of PCA by applying it to the MNIST dataset. Listing 9.12 extracts the first two principal components for the 784 long vectors describing images of digits. That is, each image, is described only by two coordinates. In doing so, we create Figure 9.9 which hints at some interesting patterns. We plot the principal component clouds for certain combinations of digits together (there are about 6,000 points for each digit. It is evident that even with two principal components, separation between certain digits is possible (e.g. 8 and 9).

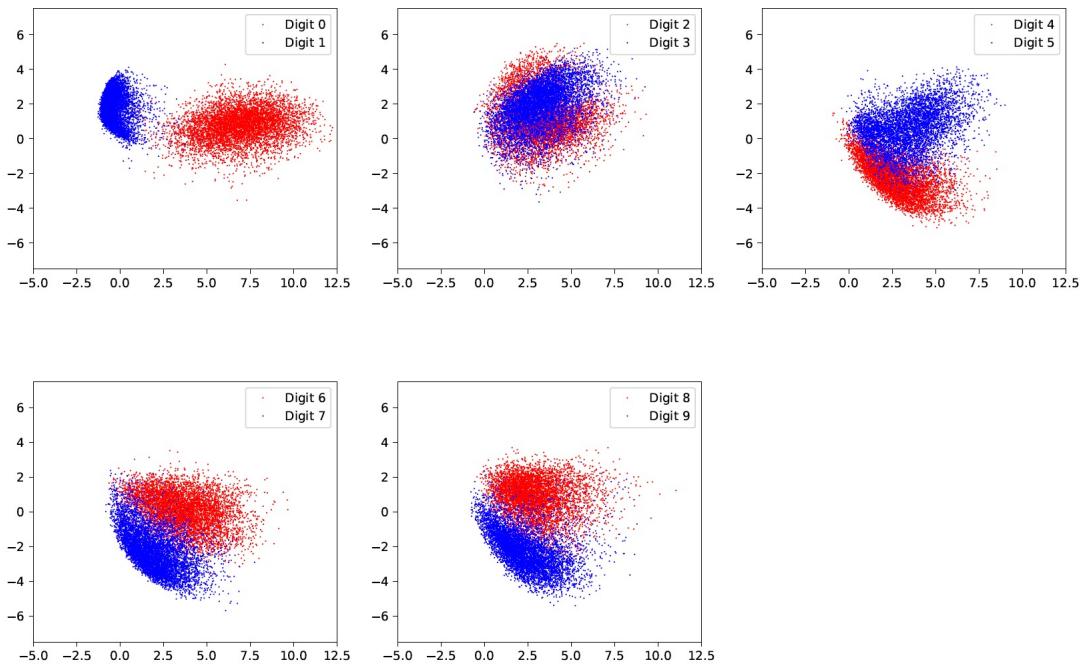


Figure 9.9: PCA on MNIST. As is evident, it is easy to separate the digits 8 and 9, however digits 2 and 3 are much harder to separate using only two principal components.

Listing 9.12: Principal component analysis on MNIST

```

1  using MultivariateStats, RDatasets, PyPlot, LinearAlgebra, Flux.Data.MNIST
2
3  imgs, labels = MNIST.images(), MNIST.labels()
4  x = hcat([vcat(float.(im)...) for im in imgs]...)
5  pca = fit(PCA, x; maxoutdim=2)
6  M = projection(pca)
7
8  function compareDigits(dA,dB)
9      imA, imB = imgs[labels .== dA], imgs[labels .== dB]
10     xA = hcat([vcat(float.(im)...)for im in imA]...)
11     xB = hcat([vcat(float.(im)...)for im in imB]...)
12     zA, zB = M'*xA, M'*xB
13     plot(zA[1,:],zA[2,:],"r.",ms="0.5",label="Digit $(dA)")
14     plot(zB[1,:],zB[2,:],"b.",ms="0.5",label="Digit $(dB)")
15     xlim(-5,12.5);ylim(-7.5,7.5);
16     legend(loc="upper right")
17 end
18
19 fig = figure(figsize = (15,10))
20 for k in 1:5
21     aFig = fig[:add_subplot](2,3,k)
22     aFig[:set_aspect]("equal")
23     compareDigits(2k-2,2k-1)
24 end
```

- In lines 3- 6 we compute the first 2 principal components for the MNIST dataset.
- In lines 8-17 we create the function `compareDigits` designed to create a plot of the first two principal components associated with digits dA and dB . First the images matching those digits are filtered in line 6. Then datasets xA and xB are created. The projection matrix M is applied to the data points, yielding the collections of 2-dimensional points zA , zB . That is, each 784 dimensional vector is collapsed to 2 dimensions. The remainder of the lines in the function plot the points.
- Lines 19-23 plot the 5 sub-plots, comparing digits neighboring digits.

9.6 Reinforcement Learning and MDP

Almost all the content of this book focuses on methods of gaining information. In some cases we explore statistical inference, in others machine learning, and in other cases system performance via simulation or numerical computation. In each case the objective is to gain additional information about the system at hand. However, why do we need such information? The most common answer is that information is needed for decision making that will affect the future of the system. In certain cases the decision making process can be detached from precise details of information retrieval and inference, and this is the mode of operation implied with most methods in this book: one obtains information, and perhaps later makes decisions based on that information. Hence in general, for most methods explored in the book, decision making is implicit and not part of the demonstrated methodology.

However there are cases where we observe the system, gain information and make decisions simultaneously. This is where techniques such as *Markov Decision Processes* (MDP), *Partially Observable Markov Decision Processes* (POMDP) and *Reinforcement Learning* (RL) play a role. We now briefly explore such methods which classically fall under the area of *stochastic dynamic programming* or *stochastic optimal control*, and more recently under *artificial intelligence*. These areas are sometimes closely related to *robotics*.

As an example, consider a scenario focusing on the engagement level of a learning student. Assume that there are L levels of engagement, $1, 2, \dots, L$ where at level 1 the student is not engaged at all and at the other extreme, at level L she is maximally engaged. Our goal is to maintain engagement as high as possible over time. We do this by choosing one of two actions at any time instant. (0): “do nothing” and let the student operate independently. (1): “stimulate” the student. In general, stimulating the student has a higher tendency to increase her engagement level, however this isn’t without cost as it requires resources.

We may denote the engagement level at time t by $X(t)$, and for simplicity we assume here that at any time $X(t)$ either increases or decreases by 1. An exception exists at the extremes of $X(t) = 1$ and $X(t) = L$. In these cases the student engagement either stays the same or increases/decreases respectively by 1. The actual transition of engagement level is random, however we assume that if our action is “stimulate” then there is more likely to be an increase of engagement than if we “do nothing”.

The control problem is the problem of deciding when to “do nothing” and when to “stimulate”.

For this we formulate a *reward function* and assume that at any time t our reward is,

$$R(t) = X(t) - \kappa A(t). \quad (9.13)$$

Here κ is some positive constant and $A(t) = 0$ if the action is to “do nothing”, while $A(t) = 1$ if the action is to “stimulate”. Hence the constant κ captures our relative cost of stimulation effort in comparison to the benefit of a unit of student engagement. We see that $R(t)$ depends both on our action and the state.

The reward is accumulated over time into an *optimization objective* via,

$$\mathbb{E} \left[\sum_{t=0}^{\infty} \beta^t R(t) \right], \quad (9.14)$$

where $\beta \in (0, 1)$ and is called the *discount factor*. Through the presence of β , future rewards are discounted with a factor of β^t , indicating that in general the present is more important than the future. There can also be other types of objectives, for example *finite horizon* or *infinite horizon average reward*, however we focus on this *infinite horizon expected discounted reward* case here.

A *control policy* is embodied by the sequence of actions $\{A(t)\}_{t=0}^{\infty}$. If these actions are chosen independently of observations then it is an *open loop* control policy. However, for our purposes, things are more interesting in the *closed loop* or *feedback control* case in which at each time t we observe the state $X(t)$, or some noisy version of it. This state feedback helps us decide on $A(t)$.

We encode the effect of action on state via a family of Markovian transition probability matrices. For each action a , we set a transition probability matrix $P^{(a)}$. In our case as there are two possible actions, we have two matrices such as for example,

$$P^{(0)} = \begin{bmatrix} 1/2 & 1/2 \\ 1/2 & 0 & 1/2 \\ 1/2 & 0 & 1/2 \\ \ddots & \ddots & \ddots & \ddots \\ \ddots & \ddots & \ddots & \ddots \\ 1/2 & 0 & 1/2 \\ 1/2 & 1/2 \end{bmatrix}, \quad P^{(1)} = \begin{bmatrix} 1/4 & 3/4 \\ 1/4 & 0 & 3/4 \\ 1/4 & 0 & 3/4 \\ \ddots & \ddots & \ddots & \ddots \\ \ddots & \ddots & \ddots & \ddots \\ 1/4 & 0 & 3/4 \\ 1/4 & 3/4 \end{bmatrix}. \quad (9.15)$$

Hence for example in state 1 (first row), if we choose action 0 then the transitions follow $[1/2 \ 1/2 \ 0 \ \dots]$, whereas if we choose action 1 then the transitions follow $[1/4 \ 3/4 \ 0 \ \dots]$. That is for a given state i , choosing action a implies that the next state is distributed according to $[P_{i1}^{(a)} \ P_{i2}^{(a)} \ \dots]$.

In the case of MDP and POMDP these matrices are assumed known, however in the case of RL these matrices are unknown. The difference between MDP and POMDP is that in MDP we know exactly in which state we are in, whereas in POMDPs our observations only hint at the current state. Hence in general we can treat MDP as the basic case and POMDP and RL can be viewed as two variants of MDP. In the case of POMDP the state isn't fully observed, while in the case of RL the transition probabilities are not known.

We don't focus on POMDPs further in this book (one can follow [Litt09] for a simple tutorial), but rather we continue with an MDP example, and then explore a basic RL example. For this, we first need to understand more technical aspects of MDP.

Optimal Policies, Value Functions and Bellman Equations

The theory of MDPs (see for example [Put14] or [Ber11]) shows that under general conditions, for such time-homogenous infinite horizon discounted cost problems, it is enough to consider *stationary deterministic Markov policies*. In this case, a policy is a function mapping every state to an action. If we denote the set of all such policies by Π , then an optimal policy is an element, $\pi \in \Pi$ that maximizes the objective, (9.14) for any initial value. As such, the *value function* is a function defined as,

$$V(i) = \max_{\pi \in \Pi} \mathbb{E} \left[\sum_{t=0}^{\infty} \beta^t R(t) \mid X(0) = i \right],$$

for any initial state i . It defines the best possible total discounted reward (value) for any current situation (state i).

We often don't know the value function for a given problem, nevertheless it is a tool that helps find optimal policies. This is because the value function appears in the *Bellman equation*:

$$V(i) = \max_{a \in \mathcal{A}} \{Q(i, a)\}, \quad \text{with} \quad Q(i, a) = r(i, a) + \beta \sum_j P_{i,j}^{(a)} V(j), \quad (9.16)$$

where $r(i, a)$ is the expected reward (with cost deducted) for state i and action a , and the set \mathcal{A} is the set of possible actions. In our example, $\mathcal{A} = \{0, 1\}$ and $r(i, a)$ is based on (9.13) yielding,

$$r(i, 0) = i, \quad \text{and} \quad r(i, 1) = i - \kappa.$$

Notice that the value function $V(\cdot)$ appears in both sides of the Bellman equation. To understand the basic idea, consider first the *Q-function* in (9.16). It measures the “quality” of being in state i and applying action a . If such a state-action pair is exhibited then immediate expected reward $r(i, a)$ is obtained followed by a transition to some random state j . This happens with probability $P_{i,j}^{(a)}$, at which point the problem continues and has value $V(j)$. However, the transition is at the next time step and hence multiplication by the discount factor β presents the value in terms of the current time step.

The Bellman equation uses the *dynamic programming principle* to determine the optimal cost in terms of maximization of the Q-function by maximizing over all actions $a \in \mathcal{A}$. It yields an equation where the “unknown” is the value function, $V(\cdot)$.

Some MDP theory deals with the validity and properties of the Bellman equation. Then, much of the study of MDP deals with methods solving the Bellman equation (9.16), or analyzing properties of the solution. Observe that if we knew the value function $V(\cdot)$ or the Q-function $Q(\cdot, \cdot)$, then we would also know an optimal policy, as we would for every state and action, i and a , seek to set $\pi(i) = a^*$ where a^* is the action that maximizes the right hand side of the Bellman equation.

Basic Value Iteration for MDP

In basic MDP, when confronted with a Bellman equation, there are typically several types of methods that can be used to solve it. Solving it implies finding the value function and with it

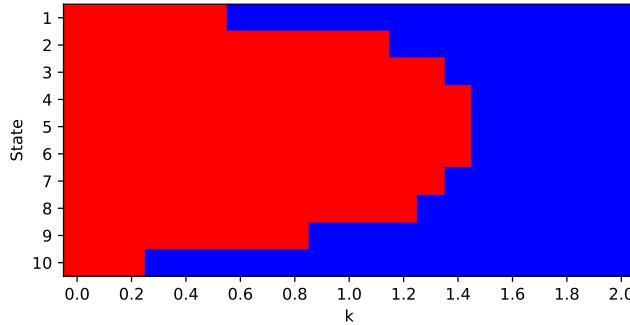


Figure 9.10: The optimal policy as a function of κ (horizontal axis) and the current state (vertical axis). Red implies to “stimulate” while blue implies “do nothing”. This is specifically the case of $L = 10$ and $\beta = 0.75$.

an optimal policy. The main known methods include *value iteration*, *policy iteration* and *linear programming*. Here we only focus on the most basic of these methods: value iteration.

Observing that the Bellman equation (9.16) contains $V(\cdot)$ both in the left hand side and the right hand side, value iteration iterates over successive value functions, $V_0(\cdot), V_1(\cdot), V_2(\cdot), \dots$, until convergence. That is we begin with some arbitrary value function, $V_0(\cdot)$ and repeatedly apply:

$$V_{t+1}(i) = \max_{a \in \mathcal{A}} \left\{ r(i, a) + \beta \sum_j P_{i,j}^{(a)} V_t(j) \right\}, \quad \text{for all } i. \quad (9.17)$$

Convergence is mathematically guaranteed (at least with finite state spaces) because the *Bellman operator*,

$$\mathcal{O}(V(\cdot)) = \max_{a \in \mathcal{A}} \left\{ r(i, a) + \beta \sum_j P_{i,j}^{(a)} V_t(j) \right\} \quad (9.18)$$

is a *contraction* in the mathematical sense. See [Put14] for more details.

Programmatically, implementing the value iteration in (9.17) is straightforward for small state space examples. We iterate and stop when the difference between iterates under some sensible *norm* is smaller than a prescribed level, ϵ . This is implemented in Listing 9.13 where we actually consider a collection of problems characterized by the cost parameter κ . For each problem, once the value function is found, we use it to determine the optimal policy. The policies are then plotted in Figure 9.10.

Listing 9.13: Value iteration for an MDP

```

1  using LinearAlgebra, PyPlot
2
3  L = 10
4  p0, p1 = 1/2, 3/4

```

```

5   beta = 0.75
6   epsilon = 0.001
7
8   function valueIteration(kappa)
9     P0 = diagm(1=>fill(p0,L-1)) + diagm(-1=>fill(1-p0,L-1))
10    P0[1,1], P0[L,L] = 1 - p0, p0
11
12    P1 = diagm(1=>fill(p1,L-1)) + diagm(-1=>fill(1-p1,L-1))
13    P1[1,1], P1[L,L] = 1 - p1, p1
14
15    R0 = collect(1:L)
16    R1 = R0 .- kappa
17
18    bellmanOperator(Vprev)=
19      max.(R0 + beta*P0*Vprev, R1 + beta*P1*Vprev)
20    optimalPolicy(V,state)=
21      (R0+beta*P0*V) [state] >= (R1+beta*P1*V) [state] ? 0 : 1
22
23    V, Vprev = fill(0,L), fill(1,L)
24    while norm(V-Vprev) > epsilon
25      Vprev = V
26      V = bellmanOperator(Vprev)
27    end
28
29    return [optimalPolicy(V,s) for s in 1:L]
30  end
31
32 kappaGrid = 0:0.1:2.0
33 policyMap = zeros(L,length(kappaGrid))
34
35 for (i,kappa) in enumerate(kappaGrid)
36   policyMap[:,i] = valueIteration(kappa)
37 end
38
39 imshow(policyMap, cmap="bwr");
40 xticks(0:2:20, 0:0.2:2); yticks(0:L-1, 1:L)
41 xlabel("k"); ylabel("State")

```

- Lines 3-6 define the model and algorithm parameters, including the discount factor `beta`, and a stopping threshold for value iteration `epsilon`.
- In lines 8-30 we implement the function `valueIteration()` which depends on a specified cost, `kappa`. The value iteration method is performed in lines 24-27, where we iterate until the normed difference between two value functions is less than or equal to `epsilon`. In line 26 we apply the Bellman operator, (9.18) via the function `bellmanOperator()` which we define in lines 18-19. Note the use of `max()` function with the broadcast dot operator (`.`), which allows us to find the element wise maximum. The function `optimalPolicy()` defined in lines 20-21 returns the optimal action to be taken given a current state. Through the use of this function along with a comprehension in line 29, `valueIteration()` returns the optimal policy for all possible states. Note that an alternative method would be to continue discovering the optimal policy during the value iteration process by considering the actions that maximize the Bellman operator. However we didn't use such an implementation here.
- The remainder of the code applies value iteration over a grid of κ values, `kappaGrid`. Note the use of `enumerate()` in the `for` loop of lines 35-37.

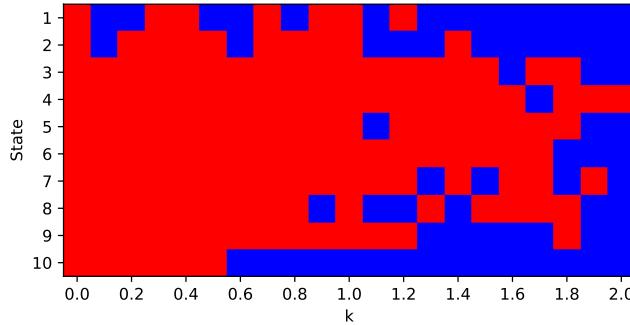


Figure 9.11: The policy learned via Q-Learning as a function of κ over long time horizons. Compare to Figure 9.10.

- In line 39, the PyPlot function `imshow()` is used to plot the optimal policies for each state, given a set κ .

Reinforcement Learning via Q-Learning

In many practical situations there isn't a clear model for the transition probability matrices $P^{(a)}$, $a \in \mathcal{A}$. For example in our engagement level example, the matrices (9.15) are a postulated model of reality. In some situations the parameters of such matrices may be estimated from previous experience, however often this isn't feasible due to changing conditions or lack of data.

Such situations are handled by reinforcement learning. The class of RL methods is a broad class of models dealing with control of systems that lack parameter knowledge. In classic control theory this situation falls under the umbrella of *adaptive control*. However in contemporary robotics, self-driving cars and artificial neural networks, RL has become the key term.

Here we explore one class of RL algorithms called *Q-learning*. The main idea of this method is to learn the Q-function as in (9.16) without explicitly decomposing $Q(i, a)$ into P , V and r . Observe from the Bellman equation, that if we were to know $Q(i, a)$ for every state i and action a , then we can also compute the optimal policy by selecting the action a that maximizes $Q(i, a)$ for every state i .

The key of Q-learning is to continuously learn $Q(\cdot, \cdot)$ while using the learned estimates to select actions as we go. For this, denote by $\hat{Q}_t(\cdot, \cdot)$ the estimate we have at time t . At any given time we attempt to balance *exploration and exploitation*. With a high probability, we decide on action a that maximizes $\hat{Q}_t(i, a)$ - this is exploitation. However, we leave some possibility to explore other actions, and occasionally decide on an arbitrary (random) action a - this is exploration. In our example, as time progresses we reduce the probability of exploration as time evolves. For example,

we use $t^{-0.2}$ for this probability, which implies that as time evolves we slowly explore less and less.

As we operate our system with Q-learning, after an action is chosen, reward r is obtained and the system transitions from state i to state j . At that point we update the (i, a) entry of the Q-function estimate as follows:

$$\hat{Q}_{t+1}(i, a) = (1 - \alpha_t) \hat{Q}_t(i, a) + \alpha_t \left(r + \beta \max_{a \in \mathcal{A}_s} \hat{Q}_t(j, a) \right). \quad (9.19)$$

Here α_t is a decaying (or constant) sequence of probabilities (in the example below we use $\alpha_t = t^{-0.2}$). The key of the *Q-learning update equation* (9.19) is a weighted average of the previous estimate $\hat{Q}_t(i, a)$ and a single sample of the right hand side of the Bellman equation (9.16). Miraculously as the system progresses under such a control, this scheme is able to estimate the Q-function and hence control the system well.

Note that ideally we would set $\{\alpha_t\}_{t=1}^{\infty}$ to satisfy,

$$\sum_{t=1}^{\infty} \alpha_t = \infty, \quad \text{and} \quad \sum_{t=1}^{\infty} \alpha_t^2 < \infty.$$

With such a condition, based on the theory of *stochastic approximation*, it is guaranteed that as $t \rightarrow \infty$, $\hat{Q}_t(i, a) \rightarrow Q(i, a)$. This property shows that (at least in principle), systems controlled via Q-learning may still be controlled in an asymptotically optimal manner, even without explicit knowledge of the underlying transition matrices $P^{(a)}$.

In Listing 9.14 below we simulate the engagement level model under Q-learning using the same parameters as before. Just as in the previous value iteration example of Listing 9.13, we do so for a range of cost parameters κ . The resulting policy is presented in Figure 9.11 which can be compared to Figure 9.10, with 10^6 time steps used for each value of κ . The control policies obtained (one for every κ) are similar to the optimal policies in Figure 9.10, but not identical. This is because (for this example) the difference between $Q(i, 0)$ and $Q(i, 1)$ is negligible for many values of i .

Listing 9.14: A Q-Learning example

```

1  using LinearAlgebra, StatsBase, PyPlot, Random
2
3  L = 10
4  p0, p1 = 1/2, 3/4
5  beta = 0.75
6  pExplore(t) = t^-0.2
7  alpha(t) = t^-0.2
8  T = 10^6
9
10 Random.seed!(1)
11
12 function QlearnSim(kappa)
13     P0 = diagm(1=>fill(p0, L-1)) + diagm(-1=>fill(1-p0, L-1))
14     P0[1,1], P0[L,L] = 1 - p0, p0
15
16     P1 = diagm(1=>fill(p1, L-1)) + diagm(-1=>fill(1-p1, L-1))
17     P1[1,1], P1[L,L] = 1 - p1, p1
18
19     R0 = collect(1:L)
20     R1 = R0 .- kappa

```

```

21
22     nextState(s,a) =
23         a == 0 ? sample(1:L,weights(P0[s,:])) : sample(1:L,weights(P1[s,:]))
24
25     Q = zeros(L,2)
26     s = 1
27     optimalAction(s) = Q[s,1] >= Q[s,2] ? 0 : 1
28     for t in 1:T
29         if rand() < pExplore(t)
30             a = rand([0,1])
31         else
32             a = optimalAction(s)
33         end
34         sNew = nextState(s,a)
35         r = a == 0 ? R0[sNew] : R1[sNew]
36         Q[s,a+1]=(1-alpha(t))*Q[s,a+1]+alpha(t)*(r+beta*max(Q[sNew,1],Q[sNew,2]))
37         s = sNew
38     end
39     [optimalAction(s) for s in 1:L]
40 end
41
42 kappaGrid = 0.0:0.1:2.0
43 policyMap = zeros(L,length(kappaGrid))
44
45 for (i,kappa) in enumerate(kappaGrid)
46     policyMap[:,i] = QlearnSim(kappa)
47 end
48
49 imshow(policyMap, cmap="bwr")
50 xticks(0:2:20, 0:0.2:2); yticks(0:L-1, 1:L)
51 xlabel("κ"); ylabel("State")

```

- In lines 3-8 we set the basic parameters as well as the functions `pExplore()` and `alpha()`, which are used for the probability of exploration and α_t respectively.
- In lines 12-40 we implement the function `QlearnSim()`, which simulates the system controlled via Q-learning. The main simulation loop is in lines 28-38. Here we choose a random action (either 0 or 1) with probability `pExplore()`, or otherwise we use the *Q-table*, `Q[]` to select an `optimalAction()`. Then line 36 updates the Q-table as per the Q-learning update equation (9.19). Note that indexation into actions in the Q-table is via 1 and 2 as Julia arrays begin with index 1), and since our action space is $\{0,1\}$, `a+1` is used in line 36.
- The remainder of the code is similar to the previous Listing 9.13.

9.7 A Taste of Generational Adversarial Networks

This section is under construction.

Chapter 10

Simulation of Dynamic Models - DRAFT

Most of the statistical methods presented in the previous chapters deal with inherently static data. There was rarely a time component involved, and typically observed random variables or vectors were assumed independent. We now move on to a different setting that involves a time component and/or dependent random variables. In general, such models are called “dynamic” as they describe change over time or space. A consequence of dynamic behavior is dependence between random variables at different points in time or space.

Our focus in this chapter is not on statistical inference for such models, but rather on model construction, simulation, analysis and control. Understanding the basics that we present here can help readers understand more complex systems and examples from *applied probability*, *stochastic operations research* and methods of *stochastic control* such as *reinforcement learning*. Dynamic stochastic models is a vast and exciting area, and here we only touch the tip of the iceberg.

A basic paradigm is as follows: in discrete time, $t = 0, 1, 2, \dots$ one way to describe a random dynamical system is via the recursion,

$$X(t+1) = f(X(t), \xi(t)), \quad (10.1)$$

where $X(t)$ is the *state* of the system at time t , $\xi(t)$ is some random perturbation and $f(\cdot, \cdot)$ is a function that yields the next state as a function of the current state and the noise component. Continuous time and other generalizations also exist. Simulation of such a dynamic model then refers to the act of using Monte Carlo to generate trajectories,

$$X(0), X(1), X(2), \dots,$$

for the purpose of evaluating performance and deciding on good control methods.

Our focus is on a few elementary cases. In Section 10.1 we consider deterministic dynamical systems. In Section 10.2 we discuss simulation of Markov Chains both in discrete time and continuous time. In Section 10.3 we discuss discrete event simulation, which is a general method for simulating processes that are subject to changes over discrete time points. In Section 10.4 we discuss models with additive noise and present a simple case of the Kalman filter. Then in Section 10.5 we briefly discuss network reliability and touch on elementary examples from *reliability theory*. We close with a discussion of common random numbers in Section 10.6. Our Monte Carlo implementation strategy

has been used in quite a few examples throughout our book, and our purpose here is to understand it a bit better.

10.1 Deterministic Dynamical Systems

Before we consider systems such as (10.1), we first consider systems without a noise component. In discrete time these can be described via the *difference equation*

$$X(t+1) = f(X(t)), \quad (10.2)$$

and in continuous time via the *Ordinary Differential Equation* (ODE),

$$\frac{d}{dt}X(t) = f(X(t)). \quad (10.3)$$

These are generally called *dynamical systems* as they describe the evolution of the “dynamic” *state* $X(t)$ over time. Many physical, biological and social systems may be modelled in this way, and a common objective is to obtain the *trajectory* of the system over time, given an *initial state* $X(0)$. In the case of a difference equation this is straightforward via recursion of equation (10.2). In continuous time we use ODE solution techniques to find the solutions of (10.3).

Discrete Time

The state $X(t)$ can take different forms. In some cases it is a scalar, in other cases a vector, and yet in other cases it is an element from an arbitrary set. As a first example, assume that it is a two dimensional vector representing normalized quantities of animals living in a competitive environment. Here $X_1(t)$ is the number of “prey” animals and $X_2(t)$ is the number of “predators”. The species then affect each other via natural growth, natural mortality, and hunting of the prey by the predators.

One very common model for such a population is the *predator prey model*, described by the *Lotka-Volterra equations*:

$$X_1(t+1) = aX_1(t)(1 - X_1(t)) - X_1(t)X_2(t), \quad (10.4)$$

$$X_2(t+1) = -cX_2(t) + dX_1(t)X_2(t). \quad (10.5)$$

Here a , c and d are positive constants that parameterize the evolution of this model. For parameter values in a certain range, there exists an *equilibrium point* to the model. For example if $a = 2$, $c = 1$ and $d = 5$ an equilibrium point is obtained via,

$$X^* = (X_1^*, X_2^*) = \left(\frac{1+c}{d}, \frac{d(a-1) - a(c+1)}{d} \right) = (0.4, 0.2).$$

To see that this is an equilibrium point, observe that using X^* for both $X(t)$ and $X(t+1)$ in (10.4) and (10.5) satisfies the equations. Hence, according to the model, once the predator and prey populations reach this point they will never move away from it. This is the definition of an equilibrium point.

Listing 10.1 below simulates the trajectory of the predator prey model by carrying out straight forward iteration over (10.4) and (10.5) given an initial state, and specific values of a , c and d . The trajectory can be seen in Figure 10.1, along with the equilibrium point.

Listing 10.1: Trajectory of a predator prey model

```

1  using PyPlot
2
3  a, c, d = 2, 1, 5
4  next(x,y) = [a*x*(1-x) - x*y, -c*y + d*x*y]
5  equibPoint = [(1+c)/d, (d*(a-1)-a*(1+c))/d]
6
7  initX = [0.8,0.05]
8  tEnd = 100;
9
10 traj = [[] for _ in 1:tEnd]
11 traj[1] = initX
12
13 for t in 2:tEnd
14     traj[t] = next(traj[t-1]...)
15 end
16
17 plot(first.(traj),last.(traj),"b.--",label="Model trajectory")
18 plot(traj[1][1], traj[1][2], "k.", ms=15, label="Initial state")
19 plot(equibPoint[1],equibPoint[2],"r+",mew="4",ms="10",label="Equilibrium point")
20 xlabel("X1"); ylabel("X2"); legend(loc="upper right")

```

- In line 4 we define the function `next()` that implements the recursion of (10.4) and (10.5).
- In line 5 the analytic equilibrium point is calculated.
- The initial state of the system is set in line 7, and the total number of discrete time points to iterate over is set in line 8.
- In line 10 we pre-allocate an array of arrays of length `tEnd`, where each sub-array is an array of two elements (representing values of X_1 and X_2 respectively). The first element of the array is then initialized in line 11.
- Lines 13-15 loop over the time horizon and the `next()` function is applied at each time to obtain the state evolution. Note the use of the splat operator `...` in line 14 for transforming the two elements of `traj[t-1]` as input arguments to `next()`.
- The remainder of the code plots Figure 10.1.

Continuous Time

We now look at the continuous time case through a physical example. Consider a block of mass M which rests on a flat surface. A spring horizontally connects the block to a nearby wall. The block is then horizontally displaced a distance z from its equilibrium position and then released. Figure 10.2 illustrates this scenario. The question is then how to describe the state of this system over time.

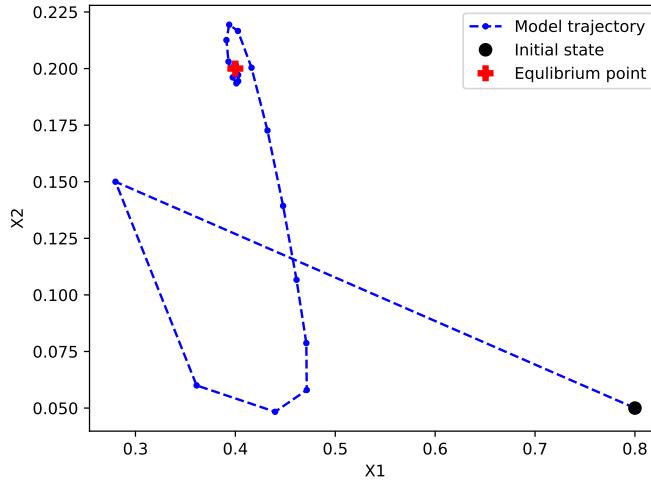


Figure 10.1: Trajectory of the predator prey model of (10.4) and (10.5) for $X_1 = 0.8$ (prey), $X_2 = 0.05$ (predator), and parameters $a, c, d = 2, 1, 5$. Given these values the system converges to the equilibrium point in red.

For this example we first make several assumptions. We assume that the spring operates elastically, and therefore the force generated by the spring is given by

$$F_s = -kz,$$

where k is the spring constant of the particular spring, and z is the displacement of the spring from its equilibrium position. Note that the force acts in the opposite direction of the displacement. In addition, we assume that dry friction exists between the block and the surface it rests on, therefore the frictional force is given by

$$F_f = -bV,$$

where b is the coefficient of friction between the block and the surface, and V is the velocity of the block. Again note that the frictional force acts in the opposite direction of the force applied, as it resists motion.

With these established we can now describe the system. Let $X_1(t)$ denote the location of the mass and $X_2(t)$ the velocity of the mass. Using basic dynamics, these can then be described via,

$$\begin{bmatrix} \dot{X}_1(t) \\ \dot{X}_2(t) \end{bmatrix} = A \begin{bmatrix} X_1(t) \\ X_2(t) \end{bmatrix} \quad \text{where} \quad A = \begin{bmatrix} 0 & 1 \\ -\frac{k}{M} & -\frac{b}{M} \end{bmatrix}. \quad (10.6)$$

The first equation of (10.6) simply indicates that $X_2(t)$ is the derivative of $X_1(t)$. The second equation can be read as,

$$M\dot{X}_2(t) = F_s + F_f.$$

Here the right hand side is the sum of the forces described above and the left hand side is “mass multiplied by acceleration”.

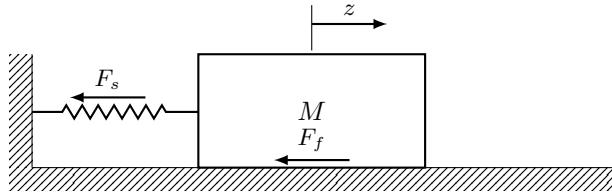


Figure 10.2: Mass and spring system, with spring force F_s , friction force F_f and applied displacement z .

With such an ODE (sometimes called a linear system of ODEs), it turns out that given initial conditions $X(0)$, a solution to this ODE is,

$$X(t) = e^{At} X(0), \quad (10.7)$$

where e^{At} is a *matrix exponential*. Hence using the matrix exponential is one way of obtaining solutions to the trajectory of $X(t)$. Many other alternative methods are implemented in Julia's DifferentialEquations package. We use both approaches in Listing 10.2 where we compute the evolution of this system given a starting velocity of zero, and a displacement of 8 units to the right of the equilibrium point. The changing state of the system is shown in the resulting Figure 10.3.

Listing 10.2: Trajectory of a spring and mass system

```

1  using DifferentialEquations, PyPlot, LinearAlgebra
2
3  k, b, M = 1.2, 0.3, 2.0
4  A = [0 1;
5      -k/M -b/M]
6
7  initX = [8.,0.0]
8  tEnd = 50.0;
9  tRange = 0:0.1:tEnd
10
11 manualSol = [exp(A*t)*initX for t in tRange]
12
13 linearRHS(x,Amat,t) = Amat*x
14 prob = ODEProblem(linearRHS, initX, (0,tEnd), A)
15 sol = solve(prob)
16
17 figure(figsize=(10,5))
18 subplot(121)
19 xlim(-7,9); ylim(-9,7)
20 plot(first.(manualSol),last.(manualSol),"b", label="Manual trajectory")
21 plot(first.(sol.u),last.(sol.u),"r.", label="DiffEq package")
22 plot(initX[1],initX[2],"k.",ms="15",label="Initial state")
23 xlabel("Displacement"); ylabel("Velocity"); legend(loc="upper right")
24
25 subplot(122)
26 plot(tRange,first.(manualSol),"b", label="Manual trajectory")
27 plot(sol.t,first.(sol.u),"r.", label="DiffEq package")
28 plot(0,initX[1],"k.",ms="15",label="Initial state");
29 xlabel("Time"); ylabel("Displacement"); legend(loc="upper right")

```

- In line 3 we set the values for the spring constant k , the friction constant b and the mass M .

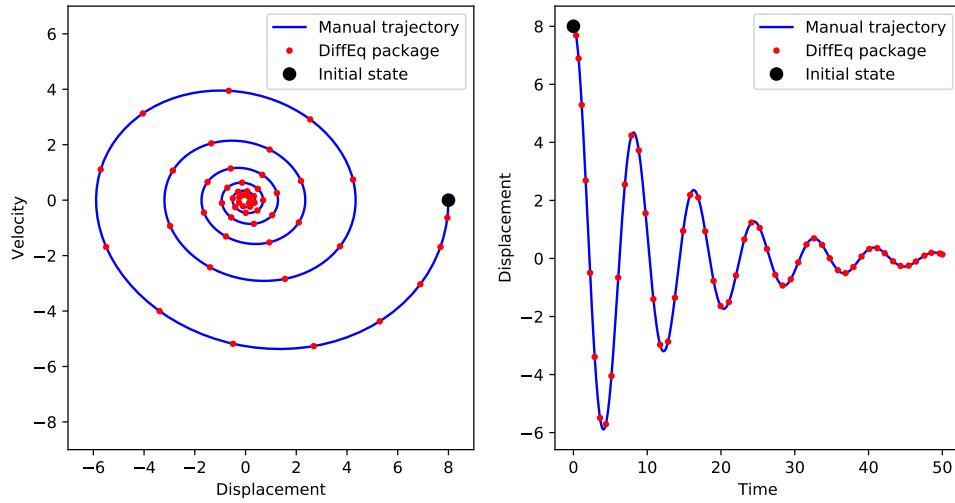


Figure 10.3: Trajectory of a spring and mass system.

- In line 4 the matrix A is defined as in (10.6).
- In line 7 the initial conditions of the system are set, with the mass displaced 8 units to the right of the equilibrium point and the velocity set to zero.
- In line 11 we compute the trajectory of the system via the brute-force approach of (10.7). Here we use `exp()` from the `LinearAlgebra` package to evaluate the matrix exponential in (10.7). The resulting array `manualSol` is an array of two dimensional arrays (state vectors), one for each point in time in `tRange`.
- In lines 13-15 the `DifferentialEquations` package is used to solve the ODE. In line 13 a function which is the right hand side of the ODE of (10.6) is defined. Line 14 defines an `ODEProblem` object as `prob`. This object is defined by the right hand side function `linearRHS`, the initial condition `initX`, a tuple of a time horizon `(0, tEnd)`, and a parameter to pass to the right hand side function, `A`. Finally line 15 uses `solve()` from the `DifferentialEquations` package to obtain a numerical solution of the ODE.
- The remaining code generates Figure 10.2, which shows the manual solution of the trajectory in blue, and discrete points along the trajectory obtained by `DifferentialEquations` in red. Observe that in line 21, `sol.u` is used to get an array of the trajectory of state from the ODE solution. Similarly, in line 27 `sol.t` is used to get the time points matching `sol.u`.

10.2 Markov Chains

In the previous section we considered systems that evolve deterministically. However sometimes it is more natural and applicable to model systems as though they have a built-in stochastic component. We now introduce and explore one such broad class of models which fall under the name of *Markov chains*. We first consider discrete time models and then move onto continuous time.

With a rich enough state space, many natural phenomena can be described via Markov chain models. Further, in certain cases such models are artificially constructed as an aid for computation. We saw such a use of Monte Carlo Markov chains (MCMC) in Section 5.7, and also briefly considered simulation of a simple discrete time Markov chain in Listing 1.7 of Section 1.3. We now dive into further details.

The basic model evolution introduced in the previous section followed $X(t+1) = f(X(t))$ where $X(t)$ is the state. That is, the next state is a direct deterministic function of the current state. Markov chains behave similarly, however in the case of a Markov chain $X(t+1)$ depends on $X(t)$ probabilistically. That is, the next state $X(t+1)$ is drawn randomly, based on a probability distribution that depends on the value of $X(t)$. For this, the model specification is typically based on a *probability transition law*,

$$p_{i,j} := \mathbb{P}(X(t+1) = j \mid X(t) = i) \quad \text{for all states } i, j. \quad (10.8)$$

Here $p_{i,j}$ specifies the probability of transitioning from a current state i to a next state j . For every i ,

$$\sum_j p_{i,j} = 1,$$

and hence the sequence $(p_{i,1}, p_{i,2}, \dots)$ specifies a probability distribution. The actual *state space* where i and j take values can vary depending on context. If the state space is countable, then the transition probabilities for all i and j describe the Markov chain. Furthermore, if the state space is finite, then the probabilities may be organized in a *transition probability matrix*, $P = [p_{i,j}]$, where each row specifies a probability distribution (or probability vector). In other cases where the state space is uncountable, it isn't possible to only consider events such as $X(t+1) = j$ and therefore the definition of (10.8) is varied slightly to allow $X(t+1) \in A$ for a rich collection of sets A . We don't discuss such situations further here, as we assume that the state space is at most countable.

At the onset of this chapter in (10.1), we specified the equation $X(t+1) = f(X(t), \xi(t))$, where $\xi(t)$ is some random perturbation. One may ask how does the evolution of a Markov chain fit this description. For this, assume that you are given the probabilities in (10.8). Now by setting the random perturbation ξ as a uniform $[0, 1]$ random variable, we are able to specify $f(i, \xi)$ as a function that evaluates the inverse CDF associated with the distribution $(p_{i,1}, p_{i,2}, \dots)$ at the point ξ . This ensures that the probabilities in (10.8) are adhered to based on the inverse probability transform (see Section 3.4). For illustration, we implement such a function $f(\cdot, \cdot)$ in Listing 10.3 below, where we specify a transition probability matrix (see the function `f1()` in the listing).

Alternatively, in certain cases it is more natural to first consider the *stochastic recursive sequence* $X(t+1) = f(X(t), \xi(t))$ and to construct the associated transition probability matrix from it as needed. For example, assume that $f(\cdot, \cdot)$ is specified as follows,

$$f(x, u) = x + u \mod 5, \quad (10.9)$$

for $x \in \{0, 1, 2, 3, 4\}$ and $u \in \{-1, 0, +1\}$. This describes a situation where the state is decremented, stays the same or incremented, all modulo 5, meaning that decrementing from 0 yields 4 and incrementing from 4 yields 0. By using this $f(\cdot, \cdot)$ in (10.1), and assuming some probability law for $\xi(t)$, we arrive at a stochastic model specifying random movement (with "wrap around") on $\{0, 1, 2, 3, 4\}$. It turns out that if we assume the noise component $\xi(t)$ is i.i.d, then such a stochastic sequence may be encoded via a transition probability matrix, and that the model is a Markov chain even though it wasn't initially specified via P .

For example, say that $\xi(t)$ takes values $\{-1, 0, +1\}$ uniformly. Then using (10.8), you may see that the corresponding transition probability matrix is

$$P = \begin{bmatrix} 1/3 & 1/3 & 0 & 0 & 1/3 \\ 1/3 & 1/3 & 1/3 & 0 & 0 \\ 0 & 1/3 & 1/3 & 1/3 & 0 \\ 0 & 0 & 1/3 & 1/3 & 1/3 \\ 1/3 & 0 & 0 & 1/3 & 1/3 \end{bmatrix}.$$

Thus we see that the dynamics of a Markov chain can be described by either a transition probability matrix, or by a stochastic recursive sequence as in (10.1). In both cases, if we specify the initial distribution $\mathbb{P}(X(0) = i)$, the evolution of the sequence of random variables, $X(0), X(1), X(2), \dots$ is well defined.

Given the Markov chain sequence $\{X(t)\}_{t=0}^{\infty}$, we are sometimes interested in its limiting statistical behavior, and at other times we use this sequence to construct another random variable and are interested in the distribution of this variable, or just in its mean. As an example, for the Markov chain described above, let τ be the minimal time such that all states have been visited:

$$\tau = \inf\{t : \exists t_0, t_1, t_2, t_3, t_4 \leq t \text{ with } X(t_i) = i\}. \quad (10.10)$$

It is clear that τ is a random quantity because depending on the realization of $\{X(t)\}_{t=0}^{\infty}$, τ may obtain different values. For example if we start with $X(0) = 0$ and then for the first 4 transitions $X(t)$ increases, then $\tau = 4$. However, it may also be that τ is a bigger number, for example if the sequence of states happens to be,

$$0, 1, 2, 1, 2, 1, 0, 4, 0, 1, 2, 1, 0, 4, 3, \dots,$$

then $\tau = 14$ because that is the first time where all states have been covered.

In Listing 10.3 we illustrate both alternatives to generating a Markov chain. The function `f1()` uses the transition probability matrix, and the function `f2()` implements (10.9) directly. For both cases we assume that $\mathbb{P}(X(0) = 0) = 1$ (i.e. we start in state 0 with certainty). We then estimate $\mathbb{E}[\tau]$ and plot estimates of the distribution of τ in Figure 10.4. It can be observed that both methods are statistically identical.

Listing 10.3: Two different ways of describing Markov chains

```

1  using LinearAlgebra, Statistics, StatsBase, PyPlot
2
3  n, N = 5, 10^6
4  P = diagm(-1 => fill(1/3,n-1),
5           0 => fill(1/3,n),
6           1 => fill(1/3,n-1))
7  P[1,n], P[n,1] = 1/3, 1/3
8
9  A = UpperTriangular(ones(n,n))
10 C = P*A
11
12 function f1(x,u)
13     for xNew in 1:n
14         if u <= C[x+1,xNew]

```

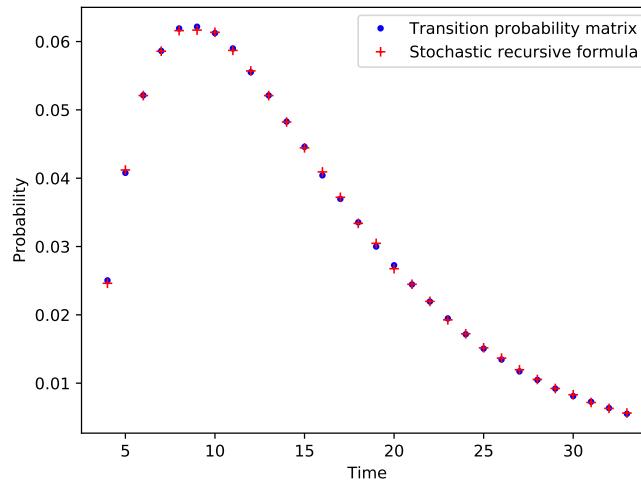


Figure 10.4: Estimates of the distribution of the time until all states in the Markov chain are visited. The blue dots are generated using the transition probability matrix, while the red dots are generated using a stochastic recursive formula.

```

15             return xNew-1
16         end
17     end
18 end
19
20 f2(x,xi) = mod(x + xi , n)
21
22 function countTau(f,rnd)
23     t = 0
24     visits = fill(false,n)
25     state = 0
26     while sum(visits) < n
27         state = f(state,rnd())
28         visits[state+1] |= true
29         t += 1
30     end
31     return t-1
32 end
33
34 data1 = [countTau(f1,rand) for _ in 1:N]
35 data2 = [countTau(f2,()>>rand([-1,0,1])) for _ in 1:N]
36 est1, est2 = mean(data1), mean(data2)
37
38 c1, c2 = counts(data1)/N,counts(data2)/N
39
40 plot(4:33,c1[1:30],"b.", label="Transition probability matrix")
41 plot(4:33,c2[1:30],"r+", label="Stochastic recursive formula")
42 xlabel("Time"); ylabel("Probability")
43 legend(loc="upper right")
44 println("Estimated mean value of tau using f1: ",est1)
45 println("Estimated mean value of tau using f2: ",est2)
46 println("\nThe matrix P:")

```

47 P

```
Estimated mean value of tau using f1: 15.0134
Estimated mean value of tau using f2: 15.00187
```

The matrix P:

```
5x5 Array{Float64,2}:
 0.333333  0.333333  0.0      0.0      0.333333
 0.333333  0.333333  0.333333  0.0      0.0
 0.0      0.333333  0.333333  0.333333  0.0
 0.0      0.0      0.333333  0.333333  0.333333
 0.333333  0.0      0.0      0.333333  0.333333
```

- In line 3 we set n as the number of states and N as the number simulation runs to carry out.
- Lines 4-7 construct the transition probability matrix P by using `diagm()` to fill the diagonals of the matrix, and by assigning values to the north-east and south-west entries as well.
- In line 9 we construct an upper triangular matrix, A and when it is right multiplied by P in line 10, we obtain a matrix of cumulative distribution vectors C.
- Lines 12-18 implement the function `f1()`. It assumes a uniform random variable u and returns a state using the inverse probability transform using the matrix C. Note that `x+1` in line 14 is because we treat the states as being `0...n` while the matrix indices are shifted by 1. For the same reason, we subtract 1 in line 15.
- Line 20 implements the function `f2()` as per (10.9).
- The function `countTau()` in lines 22-32 operates on two input arguments `f` and `rnd`, each of which is assumed a function. It then iterates (10.1) using the input arguments, and as it does so checks for the condition defining τ in (10.10). Note that we can use it with both types of $f(\cdot)$ functions, each with their respective types of random variable.
- Lines 34 and 35 exhibit calls to `countTau()` where in line 34, the input argument `f1` is augmented with the systems `rand` function, and in line 35 we create an anonymous function, `()->rand([-1, 0, 1])` as a second input argument.
- Lines 40-44 produce Figure 10.4, along with textual output that shows that both methods estimate $\mathbb{E}[\tau]$ similarly. Note that for this example it is possible to analytically show that $\mathbb{E}[\tau] = 15$.

A few more comments about discrete time Markov chains are in order. First, note that any process, $\{X(t)\}_{t=0}^{\infty}$ that satisfies this property,

$$\mathbb{P}(X(t+1) = j \mid X(t) = i, X(t-1) = i_{-1}, X(t-2) = i_{-2}, \dots) = \mathbb{P}(X(t+1) = j \mid X(t) = i). \quad (10.11)$$

is called a Markov chain. This *Markov property* indicates that given the current state ($X(t) = i$), any previous states, i_{-1}, i_{-2}, \dots do not affect the evolution of the system. This is sometimes called the *memoryless property*. Furthermore, all of the Markov chains that we consider in this chapter are *time homogenous*. This property states that for any times t_1 and t_2 ,

$$\mathbb{P}(X(t_1 + 1) = j \mid X(t_1) = i) = \mathbb{P}(X(t_2 + 1) = j \mid X(t_2) = i).$$

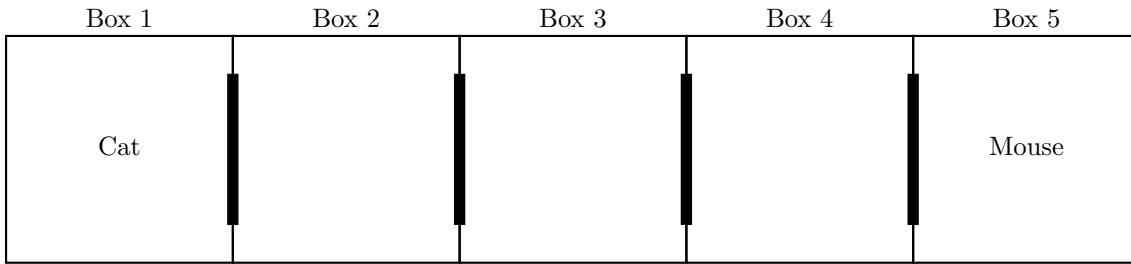


Figure 10.5: Illustration of the setup, consisting of adjacent boxes, and the starting positions of the cat and mouse, for $n = 5$ boxes.

If this were not the case, then the transition probability matrix, P would not be sufficient for describing the evolution of the Markov chain. Instead we would need a time-dependent family of matrices, $P(t)$. Also note that Markov chains possess a variety of elegant mathematical properties. See [Nor97] for an extensive introduction.

Further Discrete Time Modeling, Analysis and Simulation

Modeling using Markov chains sometimes involves constructing the state space and the associated transition probability matrix for a given scenario. In some cases this is straightforward, while in others some modeling insight is required. We now explore another example to illustrate this.

Consider the following fictional scenario. A series of boxes are connected in a row, with each adjacent box accessed via a sliding door, as in Figure 10.15. In the left most box there is a cat, and in the right most box a mouse. Then, at discrete points in time, $t = 0, 1, 2, \dots$, the doors connecting the boxes open, and both the cat and mouse migrate from their current positions, to directly adjacent boxes. They always move from their current box, randomly, with equal probability of going either left or right one box at a time.

At $t = 1$, both the cat and mouse must move to box 2 and 4 respectively. However at $t = 2$, the cat may move to either 1 or 3, and the mouse to either 3 or 5. This process of opening and closing the sliding doors repeats, until eventually the cat and mouse are in the same box, at which point the mouse is eaten by the cat and the game ends.

This situation is different from the type of Markov chain described in the previous section and from the weather chain described in Listing 1.7 of Section 1.3. In these earlier cases, the processes are *recurrent* and go on forever. In the current case the process appears to be *transient* since at a given (random) point of time, the mouse is eaten. For recurrent Markov chains typical questions often deal with the steady state stationary distribution. However in a situation such as the one we describe here, a typical question may be: how long until the mouse is eaten? As this is a random variable, we may be interested in its distribution, or at least its expected value.

When modeling such a scenario using a Markov chain there are many options because we have freedom as to how to describe the states. For example, one way is to describe the states as tuples (x, y) where x is the location of the cat and y is the location of the mouse. However, we don't have

to consider all possible combinations of x and y because it always holds that $x \leq y$. We may also observe that at any given time, both the mouse and the cat are either both in odd locations or both in even locations. This is because they are forced to move at each step, and the process alternates between odd and even. Such *periodic* phenomena can be studied further in Markov chains, however for our purposes we use this knowledge to set a small state space as follows:

State 1: (1,5). The game starts in this state. The game continues.

State 2: (2,4). The game continues.

State 3: (1,3). The game continues.

State 4: (3,5). The game continues.

State 5: (2,2), (3,3), and (4,4). The game ends.

With the states defined, we set the state space to consist of states $\{1, 2, 3, 4, 5\}$ where each state describes a situation as depicted above. From this, the stochastic matrix P is then constructed as follows:

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1/4 & 0 & 1/4 & 1/4 & 1/4 \\ 0 & 1/2 & 0 & 0 & 1/2 \\ 0 & 1/2 & 0 & 0 & 1/2 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \quad (10.12)$$

With such a representation of this Markov chain, we are now interested in the *hitting time* (i.e. the time until state 5 is reached),

$$\tau = \inf\{t : X(t) = 5\}.$$

It turns out that the theory of Markov chains goes a long way in computing expressions such as $\mathbb{E}[\tau]$. One way this can be done is by considering

$$p_0 = [1 \ 0 \ 0 \ 0], \quad \text{and} \quad T = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1/4 & 0 & 1/4 & 1/4 \\ 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 0 \end{bmatrix}.$$

Here p_0 is an initial distribution vector over the states $\{1, 2, 3, 4\}$ and T is part of the transition probability matrix P that relates to states $\{1, 2, 3, 4\}$. It can be shown using probabilistic arguments that,

$$\mathbb{E}[\tau] = p_0 (I + T + T^2 + \dots) \mathbf{1},$$

where $\mathbf{1}$ is a vector of 1's. This is done by considering all possible paths that can lead to the absorbing state 5. Here, for each $k = 0, 1, 2, \dots$, each term $p_0 T^k \mathbf{1}$ describes the probability of reaching state $\mathbf{1}$ for the first time in k steps. Now by the the *theory of non-negative matrices* it holds that

$$I + T + T^2 + \dots = (I - T)^{-1}, \quad (10.13)$$

and the inverse exists (T is a sub-stochastic matrix with maximal eigenvalue strictly inside the unit circle). This can now be computed to find the analytic solution,

$$\mathbb{E}[\tau] = p_0 (I + T + T^2 + \cdots) \mathbf{1} = p_0 (I - T)^{-1} \mathbf{1} = 4.5. \quad (10.14)$$

Hence the mean time until the cat catches the mouse is 4.5. Listing 10.4 illustrates this computation, as well as the validity of the *infinite matrix geometric series*, (10.13), sometimes called a *Leontief series*. It also shows that the maximal eigenvalue of T is in the unit circle.

Listing 10.4: Calculation of a matrix infinite geometric series

```

1  using LinearAlgebra
2
3  P = [ 0   1   0   0   0;
4      1/4 0   1/4 1/4 1/4;
5      0   1/2 0   0   1/2;
6      0   1/2 0   0   1/2;
7      0   0   0   0   1]
8
9  T = P[1:4,1:4]
10 p0 = [1 0 0 0]
11
12 for n in 1:15
13     println(first(p0*sum([T^k for k in 0:n])*ones(4)))
14 end
15
16 println("Using inverse: ", first(p0*inv(I-T)*ones(4)))
17 println("Eigenvalues of T: ", sort(eigvals(T)))

```

```

2.0
2.75
3.25
3.625
3.875
4.0625
4.1875
4.28125
4.34375
4.390625
4.421875
4.4453125
4.4609375
4.47265625
4.48046875
Using inverse: 4.5
Eigenvalues of T: [-0.707107, 0.0, 2.86229e-17, 0.707107]

```

- In line 9 we construct the matrix T as the sub-matrix of the matrix P .
- In lines 12-14 we consider the LHS series in (10.13) for increasing values of n .
- In line 16 the RHS of (10.14) is calculated. Note the use of the `inv()` function to calculate the inverse of $I-T$.
- Line 17 prints the sorted eigenvalues, and shows that the largest eigenvalue is less than 1 and hence lie in the unit circle.

In Listing 10.5 we arrive at the same result via alternative methods. One method is via a first principles implementation of the scenario, which is done in function `cmHitTime()`. The two other alternative methods make use of the `mcTraj()` function. This is a much more generic function, which creates a trajectory of a Markov chain with an arbitrary transition probability matrix P , given a starting state `initState`. It runs either for a duration of T , or stops when hitting state `stopState`. Note that by default `stopState = 0`, indicating the simulation only stops after T steps.

For illustration we use `mcTraj()` in two alternative ways. One way is by invoking it many times over (N) as follows: `mcTraj(P, 1, 10^6, 5)`, where P is the transition probability matrix in (10.12), the second and fourth arguments are the initial and stopping states respectively, and the third argument, 10^6 , is intended to be a high enough T such that the simulation only stops due to hitting state 5. Then averaging the lengths of all N trajectories yields an estimate of $\mathbb{E}[\tau]$.

The second way in which we use `mcTraj()` is related to the concept of *regenerative simulation*. We modify the final row of the transition probability matrix (10.12) by setting $P_{5,1} = 1$ and $P_{5,5} = 0$. This implies that once state 5 is reached, instead of the processes being absorbed in that state, it regenerates and starts afresh in state 1. In the language of Markov chains, this makes the transition probability matrix *irreducible* and hence (as it is a finite state space) *positive recurrent*. This then means that it posses a *stationary distribution* (or *limiting distribution*). It then holds that the inverse of the limiting probability of state 5 is the number of steps that are required to revisit the state. This allows us to generate one long trajectory of this Markov chain, estimate the limiting probability in state 5, and then obtain an estimate for $\mathbb{E}[\tau]$.

Listing 10.5: Markovian cat and mouse survival

```

1  using Statistics, StatsBase, PyPlot, Random, LinearAlgebra
2  Random.seed!(1)
3
4  function cmHitTime()
5      catIndex, mouseIndex, t = 1, 5, 0
6      while catIndex != mouseIndex
7          catIndex += catIndex == 1 ? 1 : rand([-1,1])
8          mouseIndex += mouseIndex == 5 ? -1 : rand([-1,1])
9          t += 1
10         end
11     return t
12 end
13
14 function mcTraj(P, initState, T, stopState=0)
15     n = size(P)[1]
16     state = initState
17     traj = [state]
18     for t in 1:T-1
19         state = sample(1:n, weights(P[state,:]))
20         push!(traj, state)
21         if state == stopState
22             break
23         end
24     end
25     return traj
26 end
27
28 N = 10^6

```

```

29
30 P = [ 0   1   0   0   0;
31      1/4 0   1/4 1/4 1/4;
32      0   1/2 0   0   1/2;
33      0   1/2 0   0   1/2;
34      0   0   0   0   1]
35
36 theor = [1 0 0 0] * (inv(I - P[1:4,1:4])*ones(4))
37 est1 = mean([cmHitTime() for _ in 1:N])
38 est2 = mean([length(mcTraj(P,1,10^6,5))-1 for _ in 1:N])
39
40 P[5,:] = [1 0 0 0 0]
41 pi5 = sum(mcTraj(P,1,N) .== 5)/N
42 est3 = 1/pi5 - 1
43
44 println("Theoretical: ", theor)
45 println("Estimate 1: ", est1)
46 println("Estimate 2: ", est2)
47 println("Estimate 3: ", est3)

```

```

Theoretical: 4.5
Estimate 1: 4.497357
Estimate 2: 4.501016
Estimate 3: 4.507305440667045

```

- In lines 4-12 we define the function `cmHitTime()` which returns a random time until the cat catches the mouse. The initial positions of the cat and mouse (`catIndex` and `mouseIndex` respectively) are set in line 5. The while loop in lines 6–10 then updates these position indexes until the `catIndex` and `mouseIndex` are the same. Note that in line 7, if the cat is in position/box 1, then it moves to box 2 with certainty (+1), else its position index is uniformly and randomly incremented either up or down by 1. A similar approach is used for the index/position of the mouse in line 8.
- In lines 14-26 we define the function `mcTraj()`. As opposed to `cmHitTime()`, this function generates a trajectory of a general finite state discrete time Markov chain. The argument matrix `P` is the transition probability matrix; the argument `initState` is an initial starting state; the argument `T` is a maximal duration of a simulation; and the argument `stopState` is an index of a state to stop on if reached before `T`. The default value of 0 specified indicates that there is no stop state because the state space is taken to be `1,...,n` (the dimension of `P`). The logic of the simulation is similar to the simulation in Listing 1.7. The key is line 19 where the `sample` function samples the next state from `1:n` based on probabilities determined by the respective row of the matrix `P`. Note that the iteration over the time horizon `1:T` can stop if the `stopState` is reached and the `break` statement of line 22 is executed.
- In lines 30-34 we define the transition probability matrix `P` as in (10.12).
- In line 36 we calculate the analytic solution to the average life expectancy of the mouse according to (10.14).
- In line 37 we use the `cmHitTime()` function to generate `N` i.i.d. random variables and compute their mean as `est1`.

- In line 38 we use the `mcTraj()` function setting a time horizon of 100 (effectively unbounded for this example) and a `stopState` of 5. We then generate trajectories and subtract 1 from their length to get a hitting time. Averaging this over N trajectories creates `est2`.
- The remainder of the code prints the estimates of the mean hitting time showing all four methods agree.

Continuous Time Markov Chains

A *continuous time Markov chain* also known as a *Markov jump process* is a stochastic process, $X(t)$ with a discrete state space operating in continuous time t , satisfying the property,

$$\mathbb{P}(X(t+s) = j \mid X(t) = i \text{ and information about } X(u) \text{ for } u < t) = \mathbb{P}(X(t+s) = j \mid X(t) = i). \quad (10.15)$$

That is, only the most recent information (at time t) affects the distribution of the process at a future time $(t+s)$. Other definitions can also be stated, however (10.15) captures the essence of the Markovian property, similar to (10.11) for discrete time Markov chains. An extensive account of continuous time Markov chains can be found in [Nor97].

While there are different ways to parameterize continuous time Markov chain models, a very common way is by using a so-called *generator matrix*. Such a square matrix, with dimension matching the number of states, has non-negative elements on the off-diagonal and non-positive diagonal values. This ensures that the sum of each row is 0. For example, for a chain with three states, a generator matrix may be:

$$Q = \begin{bmatrix} -3 & 1 & 2 \\ 1 & -2 & 1 \\ 0 & 1.5 & -1.5 \end{bmatrix}. \quad (10.16)$$

The values Q_{ij} for $i \neq j$ indicate the *intensity* of transitioning from state i to state j . In this example, since $Q_{12} = 1$ and $Q_{13} = 2$, there is an intensity of 1 for transitions from state 1 to state 2, and an intensity of 2 for transitions from state 1 to state 3. This implies that when $X(t) = 1$, during the time interval $t + \Delta$, for small Δ , there is a chance of approximately $1 \times \Delta$ for transition to state 2 and a chance of approximately $2 \times \Delta$ of transitioning to state 3. Furthermore there is a (big) chance of approximately $1 - 3 \times \Delta$ of not making a transition at all.

An attribute of continuous time Markov chains is that when $X(t) = i$, the distribution of time until a state transition occurs is exponentially distributed with parameter $-Q_{ii}$. In the case of the example above, when $X(t) = 1$ the mean duration until a state change is $1/3$. Furthermore, upon a state transition, the transition is to state j with probability $-Q_{ij}/Q_{ii}$. In addition, the target state j is independent of the duration spent in state i . These properties are central to continuous time Markov chains. See [Nor97] for more details.

We can also associate some discrete time Markov chains with the continuous time Models. One way to do this is to fix some time step Δ (not necessarily small), and define for $t = 0, 1, 2, 3, \dots$,

$$\tilde{X}(t) = X(t\Delta).$$

The discrete time process, $\tilde{X}(\cdot)$ is sometimes called the *skeleton* at time steps of Δ of the continuous time process $X(\cdot)$. It turns out that for continuous time Markov chains,

$$\mathbb{P}(X(t) = j \mid X(0) = i) = [e^{Qt}]_{ij},$$

i.e. is given by the i, j 'th entry of the matrix exponential. Hence the transition probability matrix of the discrete time Markov chain $\tilde{X}(t)$ is the matrix exponential $e^{Q\Delta}$. This hints at one way of approximately simulating a continuous time Markov chain: set Δ small and simulate a discrete time Markov chain with transition probability matrix $e^{Q\Delta}$. If Δ is small then,

$$e^{Q\Delta} \approx I + \Delta Q. \quad (10.17)$$

However, a much better algorithm exists. For this, consider another discrete time Markov chain associated with a continuous time Markov chain: the *embedded Markov chain* or *jump chain*. This is a process that samples the continuous time Markov chain only at jump times. It has a transition probability matrix P , with $P_{ii} = 0$ (as there isn't a transition from a state to itself), and for $i \neq j$, $P_{ij} = -Q_{ij}/Q_{ii}$. The well known *Gillespie algorithm*, which we call here the *Doob-Gillespie algorithm*, simulates a discrete time jump chain and stretches the intervals between the jumps by exponential random variables to yield a trajectory of the continuous time Markov chain. At each iteration of the algorithm, if we are in state i , we increment time by an exponential random variable with rate $-Q_{ii}$ and choose the next state based on P_{ij} .

In Listing 10.6 we consider a continuous time Markov chain with three states, starting with initial probability distribution $[0.4 \ 0.5 \ 0.1]$ and with generator matrix (10.16). The code determines the probability distribution of the state at time $T = 0.25$ showing that it is approximately $[0.27 \ 0.43 \ 0.3]$. This is achieved in three different ways. The first method is via the `crudeSimulation()` function, which is an inefficient simulation of a discrete time Markov chain *skeleton* with transition probability matrix $P = I + \Delta Q$, where Δ is taken as a small scalar value. The second method is via the `doobGillespie()` function, which is an implementation of the Doob-Gillespie algorithm presented above. Finally, the matrix exponential `exp()` is used as a non-Monte Carlo evaluation.

Listing 10.6: Simulation and analysis using a generator matrix

```

1  using StatsBase, Distributions, Random, LinearAlgebra
2  Random.seed!(1)
3
4  function crudeSimulation(deltaT,T,Q,initProb)
5      n = size(Q)[1]
6      Pdelta = I + Q*deltaT
7      state = sample(1:n,weights(initProb))
8      t = 0.0
9      while t < T
10         t += deltaT
11         state = sample(1:n,weights(Pdelta[state,:]))
12     end
13     return state
14 end
15
16 function doobGillespie(T,Q,initProb)
17     n = size(Q)[1]
18     Pjump = (Q-diagm(0 => diag(Q))).-/diag(Q)
19     lamVec = -diag(Q)

```

```

20     state = sample(1:n,weights(initProb))
21     sojournTime = rand(Exponential(1/lamVec[state]))
22     t = 0.0
23     while t + sojournTime < T
24         t += sojournTime
25         state = sample(1:n,weights(Pjump[state,:]))
26         sojournTime = rand(Exponential(1/lamVec[state]))
27     end
28     return state
29 end
30
31 T, N = 0.25, 10^5
32
33 Q = [-3 1 2
34      1 -2 1
35      0 1.5 -1.5]
36
37 p0 = [0.4 0.5 0.1]
38
39 crudeSimEst = counts([crudeSimulation(10^-3., T, Q, p0) for _ in 1:N])/N
40 doobGillespieEst = counts([doobGillespie(T, Q, p0) for _ in 1:N])/N
41 explicitEst = p0*exp(Q*T)
42
43 println("CrudeSim: \t\t", crudeSimEst)
44 println("Doob Gillespie Sim: \t", doobGillespieEst)
45 println("Explicit: \t\t", explicitEst)

```

```

CrudeSim:          [0.26845, 0.43054, 0.30101]
Doob Gillespie Sim: [0.26709, 0.43268, 0.30023]
Explicit:          [0.269073 0.431815 0.299112]

```

- In lines 4-14 we define the `crudeSimulation()` function, which approximately simulates a continuous time Markov chain through the implementation of (10.17).
- In lines 16-29 we define the `doobGillespie()` function which approximates the long term distribution of the state by simulating exponentially spaced discrete jumps according to the logic above.
- In line 31 we set the time horizon T and the number of repetitions N .
- In line 33 we set the generator matrix, Q .
- In line 37 we set the initial probability vector, p_0 .
- In lines 39-41 we evaluate the probability distribution of the state at time T via three alternative ways: `crudeSimEst`, `doobGillespieEst` and `explicitEst`.

A Simple Markovian Queue

We now briefly explore *queueing theory*, which is the mathematical study of queues and congestion (see for example [HB13]). This field of *stochastic operations research* and *applied probability* is full of mathematical models for modeling queues, waiting times and congestion. One of the most

basic models in the field is called the M/M/1 queue. In this model a single server (this is the “1” in the model name) serves customers from a queue, where each customer arrives according to a Poisson process and each one has independent exponential service times. The M’s in the model name indicate Poisson arrivals and exponential service times (the “M” stands for Markovian, or memoryless).

The number of customers in the system can be represented by $X(t)$, a continuous time Markov chain taking on values in the state space $\{0, 1, 2, \dots\}$. In this case the (infinite) tridiagonal generator matrix is given by:

$$Q = \begin{bmatrix} -\lambda & \lambda & & & \\ \mu & -(\lambda + \mu) & \lambda & & \\ & \mu & -(\lambda + \mu) & \lambda & \\ & & \mu & -(\lambda + \mu) & \ddots \\ & & & \ddots & \ddots \end{bmatrix}.$$

Here λ indicates the rate of arrival, changing $X(t)$ from state i to state $i + 1$ and μ indicates the rate of service, changing $X(t)$ from state i to state $i - 1$. A common important parameter is called the *offered load*,

$$\rho = \frac{\lambda}{\mu}.$$

When $\rho < 1$ the process $X(t)$ is stochastically stable, in which case there is a stationary distribution for the continuous time Markov chain with,

$$\lim_{t \rightarrow \infty} \mathbb{P}(X(t) = k) = (1 - \rho)\rho^k, \quad k = 0, 1, 2, \dots \quad (10.18)$$

As this is simply the geometric distribution (see Section 3.5), it isn’t hard to see that the steady state mean (which we denote by L) is,

$$L_{\text{M/M/1}} = \frac{\rho}{1 - \rho}. \quad (10.19)$$

In Listing 10.7 we implement a Doob-Gillespie simulation of the M/M/1 queue. First we plot a trajectory of the queue length process $X(t)$ over $t \in [0, 200]$ (see Figure 10.6). Then we simulate the queue for a long time horizon and check that the empirically observed mean queue length agrees with the analytic solution from (10.19).

Listing 10.7: M/M/1 queue simulation

```

1  using PyPlot, Distributions, Random
2  Random.seed!(1)
3
4  function simulateMM1DoobGillespie(lambda, mu, Q0, T)
5      t, Q = 0.0, Q0
6      tValues, qValues = [0.0], [Q0]
7      while t < T
8          if Q == 0
9              t += rand(Exponential(1/lambda))
10             Q = 1
11         else
12             t += rand(Exponential(1/(lambda+mu)))

```

```

13         Q += 2(rand() < lambda/(lambda+mu)) -1
14     end
15     push!(tValues,t)
16     push!(qValues,Q)
17 end
18 return [tValues, qValues]
19 end
20
21 function stichSteps(epochs,q)
22     n = length(epochs)
23     newEpochs = [ epochs[1] ]
24     newQ = [ q[1] ]
25     for i in 2:n
26         push!(newEpochs,epochs[i])
27         push!(newQ,q[i-1])
28         push!(newEpochs,epochs[i])
29         push!(newQ,q[i])
30     end
31     return [newEpochs,newQ]
32 end
33
34 lambda, mu = 0.7, 1.0
35 Tplot, Testimation = 200, 10^7
36 Q0 = 40
37
38 epochs, qValues = simulateMM1DoobGillespie(lambda, mu, Q0,Tplot)
39 epochsForPlot, qForPlot = stichSteps(epochs,qValues)
40 plot(epochsForPlot,qForPlot)
41 xlim(0,Tplot); ylim(0,50)
42
43 eL,qL = simulateMM1DoobGillespie(lambda, mu ,Q0, Testimation)
44 meanQueueLength = (eL[2:end]-eL[1:end-1])'*qL[1:end-1]/last(eL)
45 rho = lambda/mu
46 println("Estimated mean queue length: ", meanQueueLength )
47 println("Theoretical mean queue length: ", rho/(1-rho) )

```

Estimated mean queue length: 2.33569071839852
Theoretical mean queue length: 2.3333333333333333

- In lines 4–19 we define and implement the `simulateMM1DoobGillespie()` function. This function uses the Doob-Gillespie algorithm to create a trajectory of the M/M/1 queue. In contrast to the function defined in Listing 10.6, this function records the whole trajectory of the continuous time Markov chain. That is, the return value consists of `tValues` indicating times and `qValues` indicating state values (the state is held constant between times). Notice that in line 9 of the function implementation, the state sojourn time of rate λ is used at it matches state 0. Then in line 12, the state sojourn time has rate $\lambda + \mu$ and in line 13 there is a state transition either up or down, independently of the state sojourn time.
- In lines 21–32 we define the `stichSteps()` function, which creates a trajectory that can be plotted based on an array of time epochs `epochs`, and an array of queue lengths at each epoch `q`.
- The parameters of the queue and of the simulation are set in lines 34–36. Note that two separate times are set. The first, `Tplot = 200`, is used to plot a trajectory starting with `Q0`

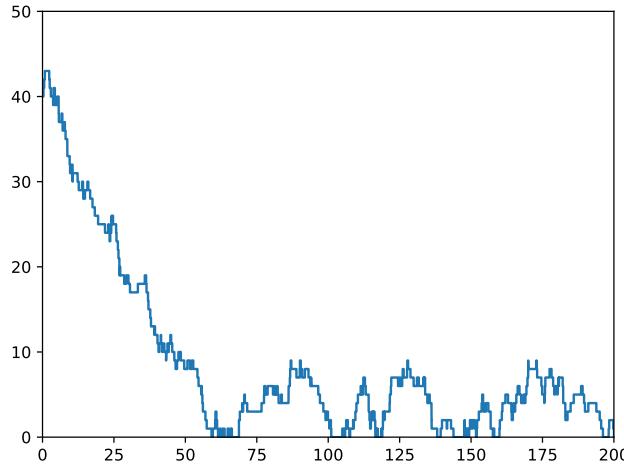


Figure 10.6: A queue length process of the M/M/1 queue starting with 40 customers in the system and with $\rho = 0.7$.

= 40 customers in the system. The second much longer duration, `Testimation`, is used for a simulation run used to estimate the mean queue length.

- In lines 38-41 the `stichSteps()` function is used to create Figure 10.6.
- In lines 43-47 the queue length estimated via simulation is compared to the theoretical queue length given by (10.19). Importantly, in line 44 the difference sequence of time jumps is calculated via `eL[2:end]-eL[1:end-1]`. By taking the inner product of this vector with the queue lengths we are able to integrate over the queue length from time 0 until the last time, `eL`, and obtain the average queue length.

10.3 Discrete Event Simulation

We now introduce the concept of *discrete event simulation*. This is a way of simulating dynamic systems that are subject to changes occurring over discrete points of time. The basic idea is to consider discrete time instances, $T_1 < T_2 < T_3 < \dots$, and assume that in between T_i and T_{i+1} the system state model $X(t)$ remains unchanged, or follows a deterministic path. At each discrete time point T_i the system state is modified due to an *event* that causes such a state change. This type of simulation is often suitable for models occurring in logistics, social service and communication.

As an illustrative hypothetical example, consider a health clinic with a waiting room for patients, two doctors operating in their own rooms and a secretary administrating patients. The state can be represented by the number of patients in the waiting room; the number of patients (say 0 or 1) speaking with the secretary; the number of patients engaged with the doctors; the activity of the doctors (say administrating aid to patients, on a break, or not engaged); and the activity of the secretary (say engaged with a patient, speaking on the phone, on a break, or not engaged).

Some of the events that may take place in such a clinic may include: a new patient arrives to the clinic, a patient enters a doctors' room, a patient leaves the doctors' room and goes to speak with the secretary, the secretary answers a phone call, the secretary completes a phone call etc. The occurrence of each event causes a state change, and these events appear over discrete time points $T_1 < T_2 < T_3 < \dots$. Hence to simulate such a health clinic, we advance simulation time, t over discrete time points.

The question then is, at which time points do events occur? The answer depends on the simulation scenario since the time of future events depends on previous events that have occurred. For example, consider the event “a patient leaves the doctors' room and goes to speak with the secretary”. This type of event will occur after the patient entered the doctors' room, and is implemented by *scheduling the event* just as the patient entered the doctors' room. That is, in a discrete event simulation, there is typically an *event schedule* that keeps track of all future events. Then, the simulation algorithm is advances time from T_i to T_{i+1} , where T_{i+1} is the time corresponding to the next event in the schedule. Hence a discrete event simulation maintains some data structure for the event schedule that is dynamically updated as simulation time progresses. General simulation packages such as AnyLogic, Arena and GoldSim do this in a generic manner, however in the examples that we present below, the event schedule is implemented in a way that is suited for our example simulation problem. For other applications, one can also look at the SimJulia package, whcih is for *process oriented* simulation in Julia, and is briefly mentioned in Section C.

We now return to the single server queue, similar to the M/M/1 queue that was simulated as a continuous time Markov chain in Section 10.2 with the Doob-Gillespie algorithm. However, in cases where inter-arrival or processing times in the queue are no longer exponentially distributed, modeling the system as a continuous time Markov chain is not easily possible (it is possible by means of extension of the state space, however this is not always the easiest implementation). Instead, simulating the system using discrete event simulation is straightforward.

In the case of a single server queue there are two types of events: (i) Customer arrives to the system, and (ii) Service completion of a customer. In this case, a discrete event simulation needs to maintain a schedule of when each of these events is to occur in the future, and we We now elaborate on this via two simple variants of the M/M/1 queue.

M/M/1 vs. M/D/1 and M/M/1/K

We now consider two variants of the M/M/1 queue model covered in 10.2, namely M/D/1 and the M/M/1/K models. In the M/D/1 model, the ‘D’ stands for deterministic service times. This is a model where there is no variability of service durations, i.e. all customers require a service of duration μ^{-1} . In a sense, such a model appears simpler than M/M/1, however mathematically it is slightly more challenging for analysis. Nevertheless, in queueing theory it is a special case of the M/G/1 queue, where ‘G’ stands for a general distribution of service time. For this, the Khinchine-Pollatzek formula (see for example [HB13]) may be used to obtain the steady state mean number of customers in a system, which exists when $\rho = \lambda/\mu < 1$.

The second M/M/1 variant that we consider, M/M/1/K, is actually mathematically simpler. This model assumes that the system has finite capacity of size K . That is, when there are $K - 1$ customers in the queue and one is being served, then any arriving customers are lost and never

return. From a mathematical perspective, this actually implies that M/M/1/K systems are finite state continuous time Markov chains with generator matrix,

$$Q = \begin{bmatrix} -\lambda & \lambda & & & \\ \mu & -(\lambda + \mu) & \lambda & & \\ & \mu & -(\lambda + \mu) & \lambda & \\ & & \ddots & \ddots & \ddots \\ & & & \mu & -(\lambda + \mu) & \lambda \\ & & & & \mu & -\mu \end{bmatrix}.$$

For any $\rho = \lambda/\mu \neq 1$ this generator matrix posses a truncated geometric steady state distribution (and for $\rho = 1$ a uniform distribution). In this case, it is easy to compute the steady state mean queue length. Hence the mean queue lengths for all three systems are given by the following formulas:

$$L_{M/M/1} = \frac{\rho}{1 - \rho}, \quad (10.20)$$

$$L_{M/D/1} = \rho \left(1 + \frac{1}{2} \frac{\rho}{1 - \rho} \right), \quad (10.21)$$

$$L_{M/M/1/K} = \frac{\rho}{1 - \rho} \frac{1 - (K+1)\rho^K + K\rho^{K+1}}{1 - \rho^{K+1}}. \quad (10.22)$$

We now compare these theoretical formulas to averages obtained via discrete event simulation. In Listing 10.8 we implement a function `queueDES()`, which performs discrete event simulation for a finite or infinite capacity queue. The simulation considers these three queue variants with $\rho \approx 0.63$, and the queue length estimates obtained for a long time horizon are shown to closely match the analytic solutions of (10.20), (10.21) and (10.22).

Listing 10.8: Discrete event simulation of queues

```

1  using Distributions, Random
2  Random.seed!(1)
3
4  function queueDES(T, arrF, serF, capacity = Inf, initQ = 0,)
5      t, q, qL = 0.0, initQ, 0.0
6
7      nextArr, nextSer = arrF(), q == 0 ? Inf : serF()
8      while t < T
9          tPrev, qPrev = t, q
10         if nextSer < nextArr
11             t = nextSer
12             q -= 1
13             if q > 0
14                 nextSer = t + serF()
15             else
16                 nextSer = Inf
17             end
18         else
19             t = nextArr
20             if q == 0
21                 nextSer = t + serF()
22             end
23             if q < capacity

```

```

24             q += 1
25         end
26         nextArr = t + arrF()
27     end
28     qL += (t - tPrev) * qPrev
29 end
30 return qL/t
31 end
32
33 lam, mu, K = 0.82, 1.3, 5
34 rho = lam/mu
35 T = 10^6
36
37 mm1Theor = rho/(1-rho)
38 md1Theor = rho*(1 + (1/2)*rho/(1-rho))
39 mm1kTheor = rho/(1-rho)*(1-(K+1)*rho^K+K*rho^(K+1))/(1-rho^(K+1))
40
41 mm1Est = queueDES(T, ()->rand(Exponential(1/lam)),
42                     ()->rand(Exponential(1/mu)))
43 md1Est = queueDES(T, ()->rand(Exponential(1/lam)),
44                     ()->1/mu)
45 mm1kEst = queueDES(T, ()->rand(Exponential(1/lam)),
46                     ()->rand(Exponential(1/mu)), K)
47
48 println("The load on the system: ", rho)
49 println("Queueing theory: ", (mm1Theor, md1Theor, mm1kTheor) )
50 println("Via simulation: ", (mm1Est, md1Est, mm1kEst) )

```

The load on the system: 0.6307692307692307
 Queueing theory: (1.708333333333333, 1.169551282051282, 1.3050346932359453)
 Via simulation: (1.7134526994574817, 1.1630297930829645, 1.302018728470463)

- In lines 4-31 we implement the function `queueDES`, which carries out a discrete event simulation queue for up to `T` time units. The arguments `arrF` and `serF` are functions that present `queueDES()` with the next inter-arrival time and next service time respectively. The argument `capacity` (with default value infinity) sets a finite queue limit to the queue (as in the M/M/1/K model). The argument `initQ` (with default value 0) is the initial queue length.
- In line 4 the initial time and queue length are set, along with the variable `qL`. This varibale is later used to calcualte the average queue length. It is essentially a running *Riemann sum*-the sum of products of the time between each event by the length of the queue in between each event, as calcualted in line 28.
- The main simulation loop is in lines 8 to 29. If the next service time occurs before the next arrival, the queue is decremented by one, and the service time is updated. If the next arrival occurs before the next service time, the queue is increased by one (as long as the queue is not at capacity) and the next arrival time is updated. Regardless of which occurs, `qL` is updated in line 28. This process continues until the time exceeds `T`. In line 30 the average queue length for the simulation is calculated and returned.
- In 33 to 35 the parameters of our three different queues are set, along with the maximum time units to be simulated `T`, and in lines 37-29 the analytic solutions of the three quues are calculated as per (10.20), (10.21), and (10.22).

- In lines 41-46 the three queues are simulated via queueDES, and the numerically estimated mean queue lengths printed alongside their analytic counterparts in lines 48-50. The results show they are in agreement.

Waiting Times in Queues

The previous example of discrete event simulation maintained the state of the queueing system only via the number of items in the queue and the scheduled events. However, in some situations it is needed to maintain a more detailed view on the system. We now consider such a case with an example of waiting times in an M/M/1 queue operating under a first come first served policy. We have already touched such a case in Listing 3.6 of Chapter 3, and in that example we implicitly used the formula,

$$\mathbb{P}(W \leq x) = 1 - \rho e^{-(\mu-\lambda)x}, \quad (10.23)$$

where W is a random variable representing the waiting time of a customer arriving to a system in steady state. This formula is obtained by considering the random variable X , representing the number of customers in the queue in steady state. As in (10.18), it has a geometric distribution. By conditioning on the values of X we are able to derive the following for strictly positive values of x :

$$\begin{aligned} \mathbb{P}(W > x) &= \sum_{k=1}^{\infty} \mathbb{P}(W > x \mid X = k) \mathbb{P}(X = k) \\ &= \sum_{k=1}^{\infty} \int_x^{\infty} f_k(u) du (1 - \rho) \rho^k \\ &= \sum_{k=1}^{\infty} \int_x^{\infty} \frac{\mu^k}{(k-1)!} u^{k-1} e^{-\mu u} du (1 - \rho) \rho^k \\ &= (1 - \rho) \lambda \int_x^{\infty} e^{-\mu u} \sum_{k=0}^{\infty} \frac{(\lambda u)^k}{k!} du \\ &= (1 - \rho) \lambda \int_x^{\infty} e^{-(\mu-\lambda)u} du \\ &= (1 - \rho) \frac{\lambda}{\mu - \lambda} e^{-(\mu-\lambda)x} \\ &= \rho e^{-(\mu-\lambda)x}. \end{aligned}$$

Note that by assuming $k = 1, 2, \dots$ customers, the waiting time of the arriving customer is distributed as the sum of k independent exponential random variables, each with mean μ^{-1} . This is the density $f_k(u)$ which is a gamma (called *Erlang*) distribution. The remainder of the calculation is straight forward.

The M/M/1 queue is special as we have an explicit formula for the distribution of the waiting time. However, if we modify the system even slightly it is often the case that such an explicit performance measure is hard to come by, and hence simulation is often used. In Listing 10.9 we carry out a simulation for the M/M/1 queue and compare it to the theoretical result from (10.23). A comparison between the ECDF and the analytic CDF is shown in Figure 10.7. It can be observed that there isn't a perfect match because we use a short time horizon in the simulation.

Listing 10.9: Discrete event simulation for M/M/1 waiting times

```

1  using DataStructures, Distributions, StatsBase, PyPlot, Random
2
3  function simMM1Wait (lambda, mu, T)
4      tNextArr = rand(Exponential(1/(lambda)))
5      tNextDep = Inf
6      t = tNextArr
7
8      waitingRoom = Queue(Float64)
9      serverBusy = false
10     waitTimes = Array{Float64,1} ()
11
12    while t < T
13        if t == tNextArr
14            if !serverBusy
15                tNextDep = t + rand(Exponential(1/mu))
16                serverBusy = true
17                push!(waitTimes, 0.0)
18            else
19                enqueue!(waitingRoom, t)
20            end
21            tNextArr = t + rand(Exponential(1/(lambda)))
22        else
23            if length(waitingRoom) == 0
24                tNextDep = Inf
25                serverBusy = false
26            else
27                tArr = dequeue!(waitingRoom)
28                waitTime = t - tArr
29                push!(waitTimes, waitTime)
30                tNextDep = t + rand(Exponential(1/mu))
31            end
32        end
33        t = min(tNextArr, tNextDep)
34    end
35
36    return waitTimes
37 end
38
39 Random.seed!(1)
40 lambda, mu = 0.8, 1.0
41 T = 10^3
42
43 data = simMM1Wait (lambda, mu, T)
44 empiricalCDF = ecdf(data)
45
46 F(x) = 1-(lambda/mu)*MathConstants.e^(-(mu-lambda)x)
47 xGrid = 0:0.1:20
48
49 plot(xGrid,F.(xGrid),"b",label="Analytic CDF of waiting time")
50 plot(xGrid,empiricalCDF(xGrid),"r",label="ECDF of waiting times")
51 xlabel("x"); xlim(0,20); ylim(0,1)
52 legend(loc="lower right");

```

- In lines 3-37 we define the main function used in this simulation, `simMM1Wait()`. This function returns a sequence of `waitTimes` for consecutive customers departing from the

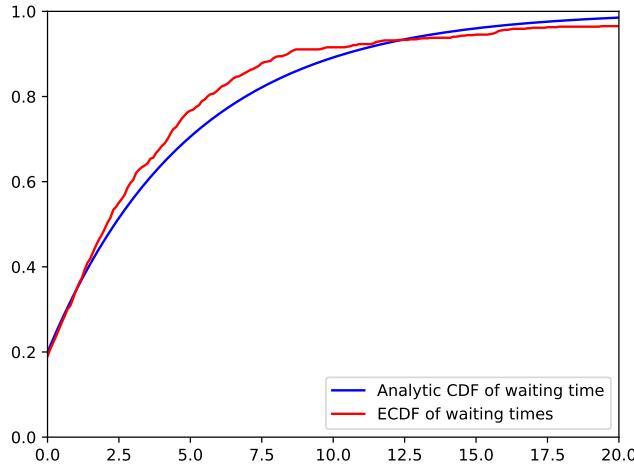


Figure 10.7: The CDF of the waiting time distribution in an M/M/1 queue with $\rho = 0.8$.

queue simulated for a time horizon T .

- The `simMM1Wait()` function uses a `Queue` data structure from the `DataStructures` package. This `waitingRoom` variable, defined in line 8, represents the waiting room of customers, and its elements represent the arrival times of customers. In line 19, when new arrivals to the busy server occur, new elements are added to `waitingRoom` via the `enqueue!()` function. In line 23, `length()` is applied to `waitingRoom` to see if the queue is empty. If it is empty, then lines 24 and 25 set the state of the system as “idle”. If the server has completed service, then the `dequeue!()` function is applied in line 27, while line 28 calculates the `waitTime`. The remainder of the code generates Figure 10.7.

10.4 Models with Additive Noise

In Section 10.1 we considered deterministic models. We then followed with inherently random models, including Markov chains and discrete event simulation. We now look at a third class of models - models that are based on deterministic models but have been modified in such a way that they now involve randomness. A basic mechanism for creating such models is to take system equations, such as (10.2), and augment them with a noise component in an additive form. Denoting the noise by $\xi(t)$ we obtain,

$$X(t+1) = f(X(t)) + \xi(t). \quad (10.24)$$

A similar type of modification can be done to continuous time systems, yielding *stochastic differential equations*. However our focus here will be on the discrete case.

As an illustrative example, we revisit the predator-prey model explored in Listing 10.1, but modify the model by adding a noise component. For this example we add i.i.d. noise with zero mean and a standard deviation of 0.02 to the prey population. This is done in Listing 10.10, and

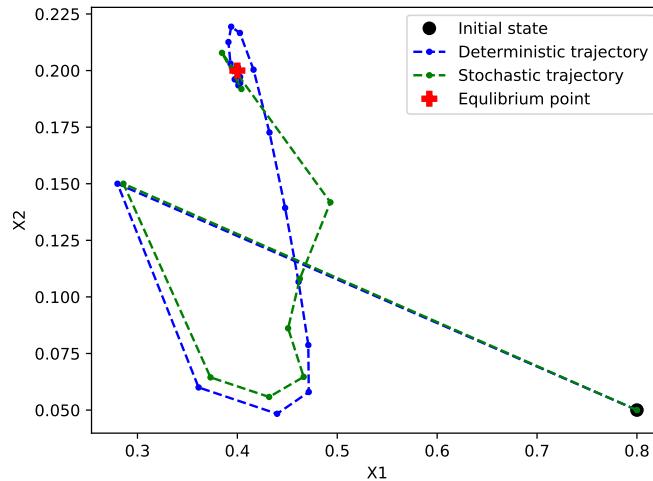


Figure 10.8: Trajectory of a stochastic predator prey model together with a deterministic model.

the resulting stochastic trajectory is then plotted alongside the previously calculated deterministic trajectory as shown in Figure 10.8. Note that this listing is very similar to Listing 10.1, with the main difference being the application of the noise vector `rand(Normal(0,sig)),0.0]`, which applies normally distributed disturbances to the prey and no explicit disturbances to the predator population.

Listing 10.10: Trajectory of a predator prey model with noise

```

1  using PyPlot, Distributions, Random
2  Random.seed!(1)
3
4  a, c, d = 2, 1, 5
5  sig = 0.02
6  next(x,y) = [a*x*(1-x) - x*y, -c*y + d*x*y]
7  equibPoint = [(1+c)/d, (d*(a-1)-a*(1+c))/d]
8
9  initX = [0.8,0.05]
10 tEnd,tEndStoch = 100, 10
11
12 traj = [[] for _ in 1:tEnd]
13 trajStoch = [[] for _ in 1:tEndStoch]
14 traj[1], trajStoch[1] = initX, initX
15
16 for t in 2:tEnd
17     traj[t] = next(traj[t-1]...)
18 end
19
20 for t in 2:tEndStoch
21     trajStoch[t] = next(trajStoch[t-1]...) + [rand(Normal(0,sig)),0.0]
22 end
23
24 plot(traj[1][1], traj[1][2], "k.", ms=15, label="Initial state")
25 plot(first.(traj),last.(traj),"b--", label="Deterministic trajectory")
26 plot(first.(trajStoch),last.(trajStoch),"g--", label="Stochastic trajectory")

```

```

27 plot(equibPoint[1], equibPoint[2], "r+", mew="4", ms="10", label="Equilibrium point")
28 xlabel("X1"); ylabel("X2"); legend(loc="upper right")
29 xlim(0,1); ylim(0,0.3)

```

State Tracking in Linear Systems

Many physical systems can be modeled by the evolution,

$$\begin{aligned} X(t+1) &= AX(t) + \xi(t), \\ Y(t) &= CX(t) + \zeta(t). \end{aligned} \tag{10.25}$$

Here $X(t)$ and $Y(t)$ are the state and observation vectors respectively, while $\xi(t)$ and $\zeta(t)$ are state and observation disturbances respectively, and are often described by independent sequences of i.i.d. random variables. Such models are often used in *control theory* and *linear system theory*. The matrix A describes the state evolution in a similar manner to the spring-mass example in (10.6), while the matrix C maps the current state to the measurement vector (prior to the addition of noise). That is, for such a system, the *sensors* are measured via $Y(t)$. In general, such systems are called *linear systems with additive noise*.

Even if the number of sensors (dimension of $Y(t)$) is much smaller than the number of state variables, (dimension of $X(t)$) we can often track the state $X(t)$. Furthermore, as we show below using *Kalman filtering*, we may even do so in the presence of the disturbance $\xi(t)$ and measurement noise $\zeta(t)$. To this end first assume that $\xi(t)$ and $\zeta(t)$ are both 0 vectors (i.e. there isn't any noise). In this case, the *Luenberger observer* is a state estimate $\hat{X}(t)$ which is parameterized by the matrix K , and operates as follows:

$$\hat{X}(t+1) = A\hat{X}(t) - K(\hat{Y}(t) - Y(t)), \quad \text{with} \quad \hat{Y}(t) = C\hat{X}(t). \tag{10.26}$$

Here, at time $t = 0$, $\hat{X}(0)$ is arbitrarily initialized, and then based on the observations $Y(0), Y(1), \dots$ the state estimate is iterated as follows:

$$\begin{aligned} \hat{X}(t+1) &= A\hat{X}(t) - K(C\hat{X}(t) - Y(t)) \\ &= (A - KC)\hat{X}(t) + KY(t). \end{aligned}$$

In this case if we consider the *estimation error*, $e(t) = X(t) - \hat{X}(t)$, then we can show that,

$$\begin{aligned} e(t+1) &= X(t+1) - \hat{X}(t+1) \\ &= AX(t) - \left(A\hat{X}(t) - K(C\hat{X}(t) - Y(t)) \right) \\ &= A(X(t) - \hat{X}(t)) - K(Y(t) - C\hat{X}(t)) \\ &= A(X(t) - \hat{X}(t)) - K(CX(t) - C\hat{X}(t)) \\ &= Ae(t) - KCe(t) \\ &= (A - KC)e(t). \end{aligned}$$

Hence if we can design a matrix K such that $A - KC$ is a stable matrix (all eigenvalues are within the unit circle), then the Luenberger observer (10.26) will have $e(t) \rightarrow 0$ as $t \rightarrow \infty$. It turns out

that if the pair A and C satisfy a rank condition called *observability* then we can always find such a matrix K , and hence always design a Luenberger observer. Then, $\hat{X}(t)$ will converge to track the system perfectly as time progresses, even if we start with an arbitrary state estimate. We don't present further details here, but instead now move onto the case with noise: Kalman filtering.

In the case of Kalman filtering, we wish to find an optimal K that will take statistical characteristics of the disturbance vectors $\xi(t)$ and $\zeta(t)$ into account (as defined in the *Linear Minimum Mean Square Error (LMMSE)* sense). Using the notation $\|\cdot\|$ for the L_2 norm, we try to set $\hat{X}(t)$ to be a linear function of the observed values which minimizes,

$$\sum_{t=0}^T \mathbb{E}[\|X(t) - \hat{X}(t)\|^2],$$

for some time horizon T , or (often more practically), the time average,

$$\lim_{T \rightarrow \infty} \frac{1}{T} \sum_{t=0}^T \mathbb{E}[\|X(t) - \hat{X}(t)\|^2],$$

which also generally equals the steady state expected mean squared error,

$$\lim_{t \rightarrow \infty} \mathbb{E}[\|X(t) - \hat{X}(t)\|^2]. \quad (10.27)$$

For these cases, the Kalman filter is a specification of the matrix K (or sequence of matrices in the case of a finite horizon) which, if used in a Luenberger observer (10.26), yields a LMMSE solution. Note that if the disturbances are assumed to be Gaussian then the LMMSE solution is also a *Minimum Mean Square Error (MMSE)* solution. That is, with Gaussian noise the Kalman filter is optimal, while with non-gaussian noise it is optimal only within the class of linear estimators.

To implement a Kalman filter, the matrix K can be computed by solving a *Riccati equation* which considers the system matrices A and C , as well as the covariance matrices of $\xi(t)$ and $\zeta(t)$. For further details, see for example [LG08]. We now present a simple scalar example of Kalman filtering.

A Scalar Example of Kalman Filtering

For this example, we construct a model based on a variation of (10.25), where all the variables are scalar and $a \in (0, 1)$:

$$\begin{aligned} X(t+1) &= aX(t) + \xi(t), \\ Y(t) &= X(t) + \zeta(t). \end{aligned} \quad (10.28)$$

This model is useful for a case where we consider the temperature of a system that reverts towards 0 by a factor of a at each time unit. If undisturbed, the temperature $X(t)$ quickly converges to 0. However, since it is subject to temperature disturbances $\xi(t)$, there are fluctuations in the temperature. This is sometimes called an *autoregressive of order 1 process*, denoted *AR(1)*. Furthermore, the measured temperature $Y(t)$ deviates from the actual temperature $X(t)$, as there are measurement disturbances present, $\zeta(t)$.

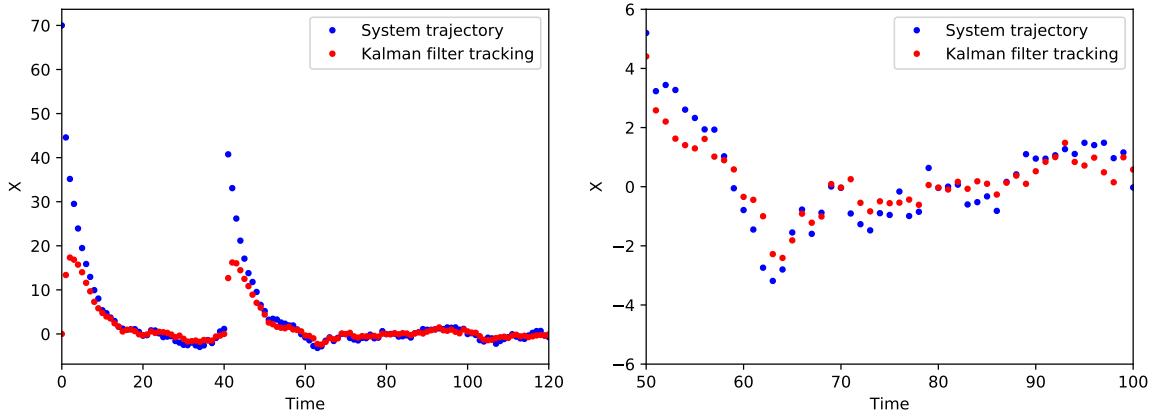


Figure 10.9: Trajectory of a linear system with noise (blue) tracked by a Kalman filter (red). At time $t = 40$ the system is disturbed and it takes a few time epochs for the Kalman filter to catch up. The right figure is a zoomed-in view of the trajectory for the time interval $50, \dots, 100$.

For this example, we now assume that $\xi(t)$ and $\zeta(t)$ are independent zero mean normal random variables with variances $\sigma_\xi^2 = 0.36$ and $\sigma_\zeta^2 = 1$ respectively, and that $a = 8$. Following (10.26), the state estimate evolution follows,

$$\hat{X}(t+1) = a\hat{X}(t) - k(\hat{X}(t) - Y(t)).$$

It turns out that the value of k that minimizes (10.27) is $k = 0.3$. This value of k takes the numerical values of a , σ_ξ^2 , and σ_ζ^2 into account.

In Listing 10.11 we carry out a simulation of this system operating for 120 time steps. Initially, the temperature is high (70 degrees) and the estimated state value is at 0. The temperature then quickly drops towards zero, and is quickly tracked by the state estimate. At time 40 an exogenous disturbance raises the temperature back up to a height of 50. While this disturbance is not part of the model, it is evident that the Kalman filter still manages to quickly respond and track the temperature. This can be seen in Figure 10.9.

Listing 10.11: Kalman filtering

```

1  using Distributions, PyPlot, Random
2  Random.seed!(1)
3
4  X0, T, a = 70.0, 120, 0.8
5  spikeTime, spikeSize = 40, 50
6  varX, varY = 0.36, 1.0
7  alpha, k = 0.8, 0.3
8
9  X, Xhat = X0, 0.0
10 xTraj, xHatTraj = [X], [Xhat]
11
12 for t in 1:T
13     global X = a*X + rand(Normal(0,sqrt(varX)))
14     global Y = X + rand(Normal(0,sqrt(varY)))
15     global Xhat = alpha*Xhat + k*(Y-Xhat)

```

```

16
17     push! (xTraj,X)
18     push! (xHatTraj,Xhat)
19
20     if t == spikeTime
21         global X += spikeSize
22     end
23 end
24
25 figure("MM vs MLE comparision", figsize=(12,4))
26 subplot(121)
27 plot(xTraj,"b.",label="System trajectory")
28 plot(xHatTraj,"r.",label="Kalman filter tracking")
29 xlabel("Time"); ylabel("X")
30 xlim(0, 120)
31 legend(loc="upper right")
32
33 subplot(122)
34 plot(xTraj,"b.",label="System trajectory")
35 plot(xHatTraj,"r.",label="Kalman filter tracking")
36 xlim(50, 100); ylim(-6,6)
37 xlabel("Time"); ylabel("X")
38 legend(loc="upper right");

```

- In line 4 the variables of the initial temperature of the system X_0 , the simulation time horizon T , and the value of a are all set.
- In line 5 we set the parameters for the unexpected additive disturbance (spike) in temperature, `spikeTime` and `spikeSize`.
- In line 6 the variance of the state disturbance (`varX`), and the variance of the measurement noise (`varY`) are defined.
- In line 7 the parameters of the Kalman filter are set.
- Lines 9-23 carry out the simulation. Line 13 is the system state evolution, and line 14 is the system observation. Line 15 implements the Kalman filter, and in lines 20–22 the spike disturbance is executed.
- The remaining lines 25-38 are used to plot Figure 10.9.

10.5 Network Reliability

We now briefly touch on the field of *network reliability* through the exploration of some simple examples. This discipline deals with the analysis of the reliability of systems which can be modeled as networks, such as roads, electric power grids, computer networks, and other systems which can be described with the aid of *graphs*. A (combinatorial) graph is a collection of *vertices* and *edges*, where the edges describe connections between vertices. For a simplistic example, one can consider a graph as a road network, where the edges represent roads and the vertices towns (such as in Figure 10.10).

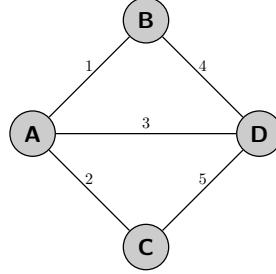


Figure 10.10: A graph with vertices $\{A, B, C, D\}$ and edges $1 = (A, B)$, $2 = (A, C)$, $3 = (A, D)$, $4 = (B, D)$, $5 = (C, D)$.

In the context of network reliability, the graph captures the relationships between the edges and vertices, and the *reliability of the network* is some statistical summary of the graph. As an example, consider the road network of Figure 10.10, and that we wish to have an active path between towns A and D . For this example there are three possible paths. However, what if the roads were subject to failure? In this case, a standard network reliability question may be: what is the probability of connectivity between towns A and D .

For our example, we consider a simplistic model, which assumes that at a snapshot of time, each road is in a failed state with probability p , independently of all other roads. Hence the reliability of the network as a function of p is,

$$r(p) = \mathbb{P}(\text{There is a path from } A \text{ to } D),$$

and in the case of our simplistic network depicted in Figure 10.10 we can actually compute $r(p)$ via,

$$\begin{aligned} r(p) &= 1 - \mathbb{P}(\text{No path from } A \text{ to } D \text{ exists}) \\ &= 1 - \mathbb{P}(A \rightarrow B \rightarrow D \text{ is broken}) \mathbb{P}(A \rightarrow D \text{ is broken}) \mathbb{P}(A \rightarrow C \rightarrow D \text{ is broken}) \\ &= 1 - (1 - (1 - p)^2) p (1 - (1 - p)^2) \\ &= 1 - p^3(p - 2)^2. \end{aligned}$$

The key in the above computation is the fact that each path does not share edges with any other path (e.g. $A \rightarrow B \rightarrow D$, or $A \rightarrow C \rightarrow D$). Therefore the individual components of each path can be put together separately. For example,

$$\mathbb{P}(A \rightarrow B \rightarrow D \text{ is broken}) = 1 - \mathbb{P}(A \rightarrow B \text{ is not broken}) \mathbb{P}(B \rightarrow D \text{ is not broken}) = 1 - (1 - p)^2.$$

Although we were able to derive an analytic equation for the reliability of this network example, it is important to note this is not typically possible. This is because as networks become more complicated, redundancy emerges, and it becomes more difficult to obtain expressions such as $1 - (1 - p)^2$ for individual paths. Hence other approaches are typically used, such as using brute-force to generate many replications of random instances of the network, and then verifying if a path exists for each.

We carry out an example of this brute-force method via Monte Carlo simulation in Listing 10.12 below. The estimates obtained are then compared with the solutions given by $r(p) = 1 - p^3(p - 2)^2$,

and the results plotted in Figure 10.11. Note that the functions defined in this code listing are not limited to the simple network of Figure 10.10, but are applicable to other networks through straightforward modifications of lines 18 and 19 (i.e. by specifying a different adjacency list, and source and destination vertices respectively).

Listing 10.12: Simple network reliability

```

1  using PyPlot, LinearAlgebra, Random
2
3  function adjMatrix(edges)
4      L = maximum(vcat(edges...))
5      aM = zeros(Int, L, L)
6      for e in edges
7          aM[ e[1], e[2] ], aM[ e[2], e[1] ] = 1, 1
8      end
9      aM
10 end
11
12 function pathExists(adjMatrix, source, destination)
13     L = size(adjMatrix)[1]
14     adjMatrix += I
15     sign.(adjMatrix^L)[source,destination]
16 end
17
18 edges = [[1,2],[1,3],[2,4],[1,4],[3,4]]
19 source, dest = 1, 4
20
21 N = 10^5
22 L = maximum(vcat(edges...))
23 adjMat = adjMatrix(edges)
24 randNet(p) = adjMat.*(rand(L,L) .<= 1-p)
25 relEst(p) = sum([pathExists(randNet(p), source, dest) for _ in 1:N]) / N
26
27 relAnalytic(p) = 1-p^3*(p-2)^2
28
29 Random.seed!(1)
30 pGrid = 0:0.02:1
31 plot(pGrid, relEst.(pGrid), "b.", label="MC")
32 plot(pGrid, relAnalytic.(pGrid), "r", label="Analytic")
33 xlim(0,1)
34 ylim(0,1)
35 xlabel("p")
36 ylabel("Reliability")
37 legend(loc="upper right");

```

- In lines 3-10 we implement the `adjMatrix()` function, which takes an array of arrays as input, and from those edges creates an adjacency matrix. In line 4, the total number of vertices of this graph (`L`) is set based on the maximal vertex appearing in `edges` via the use of `vcat()` and `...`, which are used to flatten the edges into a single array. Then in line 7, the entries of the adjacency matrix `aM` are set to 1 for each corresponding edge present.
- In lines 12-16 we implement the `pathExists()` function, which checks if there is a path between the `source` and `destination` vertices in the graph represented by the adjacency matrix `adjMatrix`. In line 14 we augment the adjacency matrix with entries on the diagonal. Then on line 15 where we raise the adjacency matrix to the `L`'th power and return the `sign`

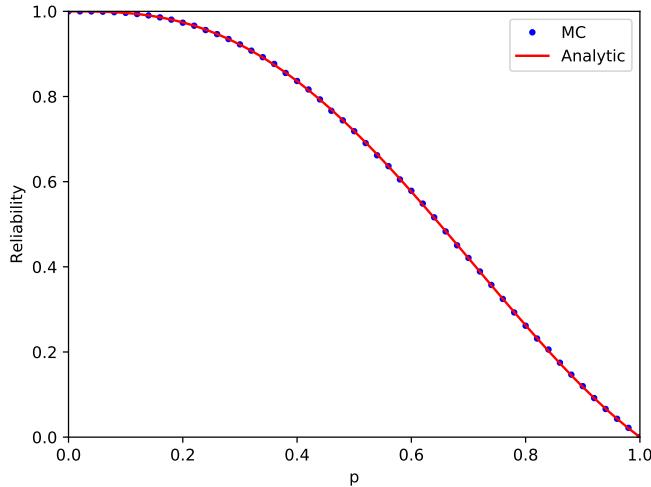


Figure 10.11: The analytic reliability function (red) compared to Monte Carlo estimates.

of the entry matching source and destination. This works because `adjMatrix` raised to the L 'th power will have a non-zero entry at i,j if and only if it is possible to reach j from i in L steps (or less, since we made sure the diagonal of `adjMatrix` has unit values).

- In line 18 we define the list of edges via the `edges` array. Note the tuples here represent node numbers (i.e. $A=1$, $B=2$, etc)
- In line 23 we set the adjacency matrix of the graph, `adjMat`.
- In line 24 we implement the `randNet()` function. It returns an adjacency matrix based on `adjMat`, but with each edge erased with probability p through the element-wise multiplication $\cdot *$ with the boolean matrix, `(rand(L,L) .<= 1-p)`. It represents the state of the network at a random snapshot in time.
- In line 25 we implement the `relEst()` function, which applies `pathExists` to `randNet(p)`, and determines if a path exists or not (given the probability of failure for each edge is p). This is repeated for N separate, independently simulated networks via a comprehension. Finally, the proportion of times a connection exists is calculated. For non-small N , `relEst()` returns an accurate reliability estimate of the failure probability for the network.
- In line 27 the analytic equation $r(p)$ is defined as `relAnalytic()`.
- The remainder of the code creates Figure 10.10, which compares the analytic solution with Monte Carlo based estimates. It can be seen that the estimated points and the analytic solution are in alignment.

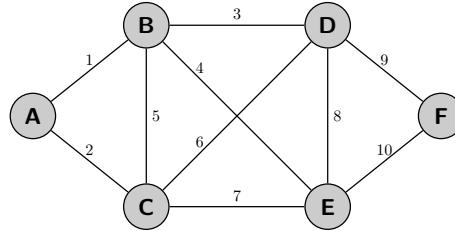


Figure 10.12: The undirected graph used in example Listing 10.13.

A Dynamic Reliability Example

We now look at a dynamic example. Instead of assuming that each edge fails with probability p , we introduce a time component, and assume that the lifetime of the edges are i.i.d. exponentially distributed random variables, with parameter λ . With such an assumption, the network state, $X(t)$ can actually be described by a continuous time Markov chain where at any given time t , $X(t)$ denotes the edge set that are yet to fail.

For our example, consider an undirected graph consisting of 6 vertices and 10 edges as shown in Figure 10.12. We now define the time until failure of this network to be when there is no longer a path from A to F. Given this, we now wish to determine the distribution and expected time until failure of the network. To do this we use the Doob-Gillespie algorithm. Note that the times between state changes of $X(t)$ are distributed exponentially, with a rate $\lambda \cdot E(X(t))$, where $E(\cdot)$ counts the number of edges in the network. For example, to begin with $X(0) = \{1, 2, \dots, 10\}$ (the full edge set of Figure 10.12), and the time until the first failure event is distributed exponentially with parameter 10λ . After the first random edge fails, the time until the next failure event is distributed exponentially with parameter 9λ , and so forth. The failure time is then the first point in time t for which the set of edges $X(t)$ does not support a path from A to F. This approach is implemented through the use of the LightGraphs package in Listing 10.13.

Listing 10.13: Dynamic network reliability

```

1  using LightGraphs, Distributions, PyPlot, StatsBase, Random
2  Random.seed!(0)
3
4  function createNetwork(edges)
5      network = Graph(maximum(vcat(edges...)))
6      for e in edges
7          add_edge!(network, e[1], e[2])
8      end
9      network
10 end
11
12 function uniformRandomEdge(network)
13     outDegrees = length.(network.fadjlist)
14     randI = sample(1:length(outDegrees),Weights(outDegrees))
15     randJ = rand(network.fadjlist[randI])
16     randI, randJ
17 end
18
19 function networkLife(network,source,dest,lambda)
20     failureNetwork = copy(network)

```

```

21      t = 0
22      while has_path(failureNetwork, source, dest)
23          t += rand(Exponential(1/(failureNetwork.ne*lambda)))
24          i, j = uniformRandomEdge(failureNetwork)
25          rem_edge!(failureNetwork, i, j)
26      end
27      t
28  end
29
30 lambda1, lambda2 = 0.5, 1.0
31 roads = [[1,2],[1,3],[2,4],[2,5],[2,3],[3,4],[3,5],[4,5],[4,6],[5,6]]
32 source, dest = 1, 6
33 network = createNetwork(roads)
34 N = 10^6
35
36 failTimes1 = [ networkLife(network,source,dest,lambda1) for _ in 1:N ]
37 failTimes2 = [ networkLife(network,source,dest,lambda2) for _ in 1:N ]
38
39 plt[:hist](failTimes1, 200, color="b", histtype = "step",
40             density=true, label="lambda = $(lambda1)")
41 plt[:hist](failTimes2, 200, color="r", histtype = "step",
42             density=true, label="lambda = $(lambda2)")
43 xlim(0,5)
44 xlabel("t")
45 legend(loc="upper right")
46
47 println("Edge Failure Rate = $(lambda1): Mean failure time = ",
48         mean(failTimes1), " days.")
49 println("Edge Failure Rate = $(lambda2): Mean failure time = ",
50         mean(failTimes2), " days.")

```

Edge Failure Rate = 0.5: Mean failure time = 1.4471182849093784 days.
 Edge Failure Rate = 1.5: Mean failure time = 0.48129663793885885 days.

- In lines 4-10 we implement the `createNetwork()` function, which creates a code `Graph` object from the `LightGraphs` package based on a list of edges. In line 5, the `Graph()` constructor is called, for which `maximum(vcat(edges...))` defines the number of vertices. Then in line 7 following a `for` loop is used to loop over each element of `edges` and add them to the graph via the `add_edge!()` function from `LightGraphs`. The return value, `network` is a `graph` object.
- In lines 12-17 we implement `uniformRandomEdge()`, which takes a graph object from `LightGraphs` and returns a random uniformly selected edge (in the form of a tuple). In line 13, `outDegrees` is set by broadcasting `length()` to each element of `network.fadjlist` (i.e. to each element of the adjacency list). This sets `outDegrees` as an array counting how many edges point out from each of the vertices. In line 14 we set `randI` to be an index of a vertex by sampling with weights based on `outDegrees`. Then line 15 sets `randJ`. In line 16, the tuple, `(randI, randJ)` is returned.
- In lines 19-28 we implement the `networkLife()` function, which takes a `network` as input, and then degrades it according to a Poisson process at rate `lambda`. At each state it checks if a connection exists between `source` and `destination`, and returns the time when a path no longer exists. First, in line 20 the `copy()` function is used to create a copy of `network` in

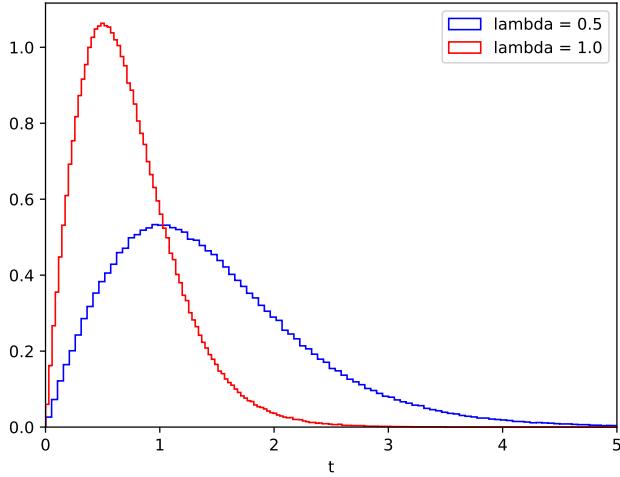


Figure 10.13: Comparison of the distribution of time until failure for $\lambda = 0.5$ and $\lambda = 1$.

place. This is because `network` is passed by reference and we wish to degrade a true copy of it, `failureNetwork`. Then in lines 22-26, the `LightGraphs` function `has_path()` is used to see if the network has a path from `source` to `dest`. Between each iteration, we wait for a duration that is exponentially distributed with a rate proportional to the number of edges (`failureNetwork.ne`). Then in line 23 `uniformRandomEdge()` is used to choose an edge, and in line 25 this is then removed via `rem_edge!()`.

- In lines 30-34 the network shown in Figure 10.12 is defined, along with the other parameters of the problem. For this example we use two different decay factors (`lambda1` and `lambda2`), and in lines 36-37 the `networkLife()` function is used along with comprehensions to generate `N` separate network failure times for each. The remaining lines are used to plot the estimated distributions of network lifetimes in Figure 10.13, as well as print the mean times until failure.

10.6 Common Random Numbers and Multiple RNGs

More than half of the examples in this book have involved some sort of (pseudo-)random number generation, often for the purpose of estimating some parameter, or performance measure. In such cases, one wishes to make the process as efficient as possible (i.e. one wishes to reduce the number of computations performed). However, there is an inherent tradeoff in this, since by reducing the number of computations one also reduces the confidence one has in the value of the parameter. Hence the concept of *variance reduction* is often employed to reduce the number of simulation runs, while maintaining the same precision of the parameter of interest. In this section we focus on one such technique: *common random numbers*.

We have actually already used this technique in several examples; see for example Listing 7.3.

In these cases, the seed was fixed via `Random.seed!()` and a parameter was varied over some desired range. In order to gain more insight into common random numbers, consider the random variable,

$$X \sim \text{Uniform}(0, 2\lambda(1 - \lambda)) \quad (10.29)$$

Clearly,

$$\mathbb{E}_\lambda[X] = \lambda(1 - \lambda).$$

Hence for this example, it is immediate that the expectation is maximized when $\lambda^* = 1/2$, which yields $\mathbb{E}_{\lambda^*}[X] = 1/4$. Now, for illustrative purposes, say we wish to find this optimal λ using simulation. Here we simulate n copies of X for each λ in some grid over $(0, 1)$, and for each λ obtain an estimate via,

$$\hat{m}(\lambda) := \widehat{\mathbb{E}_\lambda[X]} = \frac{1}{n} \sum_{i=1}^n X_i, \quad (10.30)$$

and then we choose $\widehat{\lambda^*}$ as the λ with maximal $\hat{m}(\lambda)$.

The straightforward approach to simulation would be to repeat the evaluation of $\hat{m}(\lambda)$, and to use different random values each time. This would be the behavior if `rand()` was simply used repetitively, and the seed was not modified between each evaluation. Such an approach effectively implies (assuming ideal random numbers) that for each λ , each evaluation of $\hat{m}(\lambda)$ is independent of the other evaluations. Note that each such evaluation requires generating n copies of X , with each such copy requiring a random number, N , (Poisson distributed) copies of Z_i .

The method of *common random numbers* is to use the same random numbers (i.e. stream of random numbers) for every λ over the grid. Mathematically this can be viewed as fixing an ω_0 in the probability sample space Ω (see Section 2.1) and re-evaluating the estimate $\hat{m}(\lambda, \omega_0)$ for all values of λ . The idea is motivated by the assumption that for close value of λ , say λ_0 and λ_1 , the estimate of $\hat{m}(\lambda_0, \omega_0)$ and $\hat{m}(\lambda_1, \omega_0)$ don't significantly differ.

In Listing 10.14 we consider the example of estimating the maximizer λ^* from (10.32) above, and compare estimates obtained naively using different random numbers each time with estimates obtained via the use of common random numbers. The results shown in Figure 10.14 illustrate that for estimates obtained using common random numbers, the neighbouring estimates do not differ greatly, much less variance is observed, and the estimates much closer to the true parameter values when compared to the estimates obtained that did not use common random numbers.

Listing 10.14: Variance reduction via common random numbers

```

1  using Distributions, PyPlot, Random
2
3  seed = 1
4  n = 10
5  lamGrid = 0.01:0.01:0.99
6
7  theorM(lam) = mean(Uniform(0,2*lam*(1-lam)))
8  estM(lam) = mean(rand(Uniform(0,2*lam*(1-lam)),n))
9
10 function estM(lam,seed)
11     Random.seed!(seed)
12     estM(lam)
13 end
14

```

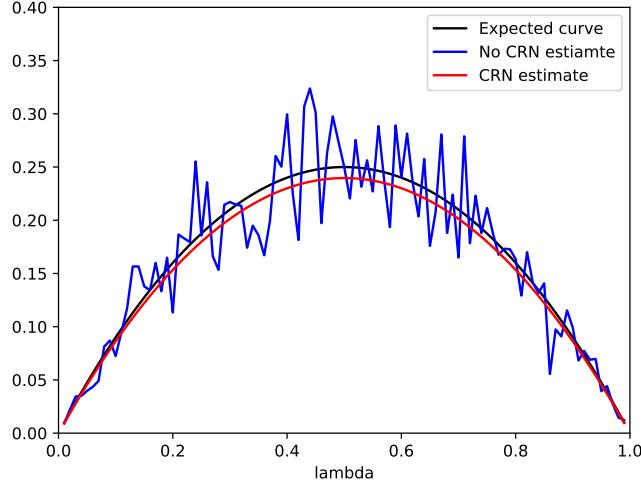


Figure 10.14: Comparrison of common random number estimates with estiamtes that did not use common random numbers. The actual expected curve is also shown.

```

15 trueM = theorM.(lamGrid)
16 estM0 = estM.(lamGrid)
17 estMCRN = estM.(lamGrid,seed)
18
19 plot(lamGrid,trueM,"k", label="Expected curve")
20 plot(lamGrid,estM0,"b", label="No CRN estiamte")
21 plot(lamGrid,estMCRN,"r", label="CRN estimate")
22 xlim(0,1); ylim(0,0.4)
23 xlabel("lambda")
24 legend(loc="upper right")

```

- In line 7 we define the function `theorM()` which returns the theoretical mean, $\lambda(1 - \lambda)$.
- In line 8 the function `estM()` is defined, which creates a sample of `n` random variables and computes their sample mean.
- In lines 10-13 we define an additional method for `estM()`. This method takes two arguments, the second one being `seed`. It sets the random seed in line 11 and then estimates the sample mean via the function of line 8.
- In line 15 the theoretical means are evaluated over the grid `lamGrid`, and the vector is set as `trueM`.
- In line 16 `estM` is used to estimate the means over `lamGrid` without the use of common random numbers.
- In line 17 the second method of `estM` is used to estiamte the means over `lamGrid` through the use of common random numbers. Here, the second argument `seed` is passed to `Random.seed!` in line 11 before the random numbers are generated (via the first method of `estM` in line 12).

This way the same stream of random numbers are used in each estimate. The remainder of the code is used to create Figure 10.14.

The Case for Using Multiple RNGs

We now consider another example, with the purpose of showing that in addition to the benefit of using common random numbers, there may sometimes be benefit from using multiple random number generators (RNGs) instead of a single RNG.

Consider the random variable,

$$X = \sum_{i=1}^N Z_i, \quad (10.31)$$

where $N \sim \text{Poisson}(K\lambda)$ and $Z_i \sim \text{Uniform}(0, 2(1-\lambda))$ with $\lambda \in (0, 1)$. In this case, it is possible to show that

$$\mathbb{E}_\lambda[X] = K\lambda(1 - \lambda).$$

In this simple example, it is easy to see that the expectation is maximized when $\lambda^* = 1/2$. In which case, $\mathbb{E}_{\lambda^*}[X] = K/4$. However for illustration purposes, say we wish to find this optimal λ using simulation. In this case we may simulate n copies of X for each λ in some grid on $(0, 1)$ and for each λ , obtain an estimate via,

$$\hat{m}(\lambda) := \widehat{\mathbb{E}_\lambda[X]} = \frac{1}{n} \sum_{i=1}^n X_i. \quad (10.32)$$

We then choose $\widehat{\lambda^*}$ as the λ with maximal $\hat{m}(\lambda)$.

The straightforward approach is to repeat the evaluation of $\hat{m}(\lambda)$, each time using different random values. This would be the behavior if `rand()` is used repetitively without modifying the seed. Such an approach effectively implies (assuming ideal random numbers) that for each λ , each evaluation of $\hat{m}(\lambda)$ is independent of the other evaluations. Note that each such evaluation requires generating n copies of X , with each such copy requiring a random number (Poisson distributed) copies of Z_i .

The method of *common random numbers* is to repeat the use of the same random numbers for every λ . Mathematically this can be viewed as fixing an ω_0 in the probability sample space Ω (see Listing 1.16) and re-evaluating the estimate $\hat{m}(\lambda, \omega_0)$ for all values λ . The idea, is motivated by the assumption that for near values of λ , say λ_0 and λ_1 the estimate of $\hat{m}(\lambda_0, \omega_0)$ and $\hat{m}(\lambda_1, \omega_0)$, with same ω_0 don't significantly differ.

As an example, momentarily modify (10.31) and consider $N = 1$. That is, in (10.32), X_i may be replaced by Z_i . We now take $n = 10$ and compare the true expected curve, $\lambda(1-\lambda)$ with an estimate $\hat{m}(\lambda)$ (no common random numbers) and $\hat{m}(\lambda, \omega_0)$ (yes common random numbers). Listing 10.14 produces Figure 10.14 where we compare the three curves. As is clearly evident, in the absence of common random numbers, the estimated curve in red, $\hat{m}(\lambda)$ is very noisy. As opposed to that, when using common random numbers, the estimated curve in blue, appears much smoother and doesn't significantly differ from the true curve (in green). Clearly, in this example, trying to estimate the maximizer λ^* using common random numbers would generally yield a much better result.

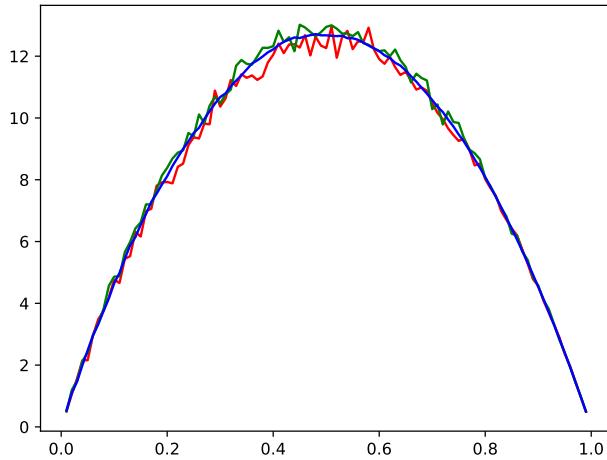


Figure 10.15: The effect of using two RNGs together with common random numbers: The blue curve is obtained with two RNGs and performs better than the red curve (single RNG and no common random numbers) and green curve (single RNG with common random numbers).

Listing 10.15: A case for two RNGs

```

1  using Distributions, PyPlot, Random
2
3  N, K, M = 10^2, 50, 10^3
4  lamRange = 0.01:0.01:0.99
5
6  prn(lambda, rng) = quantile(Poisson(lambda), rand(rng))
7  zDist(lam) = Uniform(0,2*(1-lam))
8
9  rv(lam, rng) = sum([rand(rng,zDist(lam)) for _ in 1:prn(K*lam, rng)])
10 rv2(lam, rng1, rng2) = sum([rand(rng1,zDist(lam)) for _ in 1:prn(K*lam, rng2)])
11
12 mEst(lam, rng) = mean([rv(lam, rng) for _ in 1:N])
13 mEst2(lam, rng1, rng2) = mean([rv2(lam, rng1, rng2) for _ in 1:N])
14
15 function mGraph0(seed)
16     singleRng = MersenneTwister(seed)
17     [mEst(lam, singleRng) for lam in lamRange]
18 end
19 mGraph1(seed) = [mEst(lam,MersenneTwister(seed)) for lam in lamRange];
20 mGraph2(seed1,seed2) = [mEst2(lam,MersenneTwister(seed1),
21                         MersenneTwister(seed2)) for lam in lamRange];
22
23 plot(lamRange,mGraph0(1987),"r")
24 plot(lamRange,mGraph1(1987),"g")
25 plot(lamRange,mGraph2(1987,1988),"b")
26
27 argMaxLam(graph) = lamRange[findmax(graph)[2]]
28
29 std0 = std([argMaxLam(mGraph0(seed)) for seed in 1:M])
30 std1 = std([argMaxLam(mGraph1(seed)) for seed in 1:M])

```

```

31 std2 = std([argMaxLam(mGraph2(seed, seed+M)) for seed in 1:M])
32
33 println("Standard deviation with no CRN: ", std0)
34 println("Standard deviation with CRN and single RNG: ", std1)
35 println("Standard deviation with CRN and two RNGs: ", std2)

```

```

Standard deviation with no CRN: 0.037080520020152975
Standard deviation with CRN and single RNG: 0.03411444555309958
Standard deviation with CRN and two RNGs: 0.014645353747396726

```

- In line 3 we define N , the number of repetitions to carry out for each value of λ ; the constant K ; and M , this is the number of repetitions to carry out in total for estimating the argmax.
- In line 6 we define our function `prn()`. It uses the inverse probability transform to generate a Poisson random variable with parameter `lambda` and with a random number generator `rng`.
- In line 7 we define a function for creating a $\text{uniform}(0, 2(1 - \lambda))$ distribution.
- In lines 9 and 10 we create the two central functions for this example. The function `rv()` uses a single random number generator to generate the random variable (10.31). Then the function `rv2()` achieves this with two random variables. One for the uniforms random variables and one for the Poisson random variable.
- Lines 12 and 13 create the functions `mEst()` and `mEst2()`. The first uses a single random number generator and the second uses two random number generators.
- Lines 15-21 define the functions `mGraph0()`, `mGraph1()` and `mGraph2()`.
- Lines 23-25 generate plots.
- Lines 27-35 compare performance of the argmax.

Appendix A

How-to in Julia - DRAFT

The code examples in this book are primarily designed to illustrate statistical concepts. However, they also have a secondary purpose, as they serve a way of **learning how to use Julia by example**. Towards this end, the appendix serves to link language features and uses to specific Julia code listings. This appendix can be used on an ad-hoc basis to find code examples where you can see “how to” do specific things in Julia. Once you find the specific “how to” that you are looking for, you can refer to its associated code example, referenced via “ \Rightarrow ”.

The appendix is broken up into several subsections as follows. Basics (Section A.1), deals with basic language features. Text and I/O (Section A.2) deals with textual operations as well as input and output. Data Structures (Section A.3), deals with data structures and their use. Data Frames (Section A.4) deals with Data Frames. Mathematics (Section A.5), covers various mathematical aspects of the language. Randomness, Statistics and Machine Learning (Section A.6), deals with random number generation, elementary statistics, distributions, statistical inference and machine learning. Graphics (Section A.7), deals with plotting, manipulation of figures and animation.

A.1 Basics

Types

- Check the type of an object.
 \Rightarrow [Listing 1.2](#).
- Convert the type of an array.
 \Rightarrow [Listing 1.7](#).
- Use big representation of numbers using `big()`.
 \Rightarrow [Listing 2.3](#).

Variables

- Assign two values in a single statement (using an implicit tuple).

⇒ Listing 1.5.

Conditionals and Logical Operations

- Use the conditional `if` statement.
⇒ Listing 1.5.
- Use the shorthand conditional formatting operator `? : .`
⇒ Listing 2.5.
- Carry out element-wise and using `.&.`
⇒ Listing 4.5.

Loops

- Create a while loop.
⇒ Listing 1.10.
- Loop over values in an array.
⇒ Listing 1.1.
- Create nested for loops.
⇒ Listing 1.5.
- Break out of a loop with `break`.
⇒ Listing 2.5.
- Loop over an enumeration of (Index,value) pairs created by `enumerate()`.
⇒ Listing 3.23.

Functions

- Create a function.
⇒ Listing 1.5.
- Create a one line function.
⇒ Listing 1.9.
- Create a function that returns a function.
⇒ Listing 1.6.
- Pass functions as arguments to functions.
⇒ Listing 10.8.
- Create a function with a multiple number of arguments.
⇒ Listing 1.6.
- Use an anonymous function.
⇒ Listing 1.14.

- Define a function inside another function.
⇒ [Listing 2.4](#).
- Create a function that returns a tuple.
⇒ [Listing 7.11](#).
- Setup default values to function arguments.
⇒ [Listing 10.8](#).

Other Basic Operations

- Check the running time of a block of code.
⇒ [Listing 1.3](#).
- Increment values using `+=`.
⇒ [Listing 1.7](#).
- Do element-wise comparisons such as for example using `.>`.
⇒ [Listing 2.9](#).
- Apply an element-wise computation to a tuple.
⇒ [Listing 2.10](#).
- Use the logical `xor()` function.
⇒ [Listing 2.12](#).
- Set a numerical value to be infinity with `Inf`.
⇒ [Listing 3.6](#).
- Include another block of Julia code using `include()`.
⇒ [Listing 3.34](#).
- Find the maximal value amongst several arguments using `max()`.
⇒ [Listing 7.3](#).
- Find the minimal value amongst several arguments using `min()`.
⇒ [Listing 7.4](#).

Metaprogramming

- Create a formula with `Formula()` and `Expr()`.
⇒ [Listing 8.17](#).

Interacting with Other Languages

- Copy data to the R environment with `@rput` from package `RCall`.
⇒ [Listing 1.17](#).
- Get data from the R environment with `@rget` from package `RCall`.
⇒ [Listing 1.17](#).

- Execute an R language block with the command `R` from package `RCall`.
⇒ [Listing 1.17](#).
- Setup a Python object in Julia using `@pyimport` from package `PyCall`.
⇒ [Listing 1.18](#).

A.2 Text and I/O

Strings

- Split a string based on whitespace with `split()`.
⇒ [Listing 1.8](#).
- Use `tex` formatting for strings.
⇒ [Listing 2.4](#).
- See if a string is a substring of another string with `occursin()`.
⇒ [Listing 4.16](#).

Text Output

- Print text output including new lines, and tabs.
⇒ [Listing 1.1](#).
- Format variables within strings when printing.
⇒ [Listing 2.1](#).

Reading and Writing From Files

- Open a file for writing with `open()`.
⇒ [Listing 4.7](#).
- Open a file for reading with `open()`.
⇒ [Listing 4.16](#).
- Write a string to a file with `write()`.
⇒ [Listing 4.7](#).
- Close a file after it was opened.
⇒ [Listing 4.7](#).
- Read from a file with `read()`.
⇒ [Listing 4.7](#).
- Display the current working directory with `pwd()`.
⇒ [Listing 4.17](#).
- See the list of files in a directory with `readdir()`.
⇒ [Listing 4.17](#).

CSV Files

- Read a CSV file to create a data frame with `CSV.read()`.
⇒ [Listing 4.1](#).
- Read a CSV file to create an array with `CSV.read()` without a header.
⇒ [Listing 6.1](#).
- Read a CSV file that is transposed with `transpose = true` in `CSV.read()`.
⇒ [Listing 4.6](#).
- Write to a CSV file with `CSV.write()`.
⇒ [Listing 4.5](#).
- Read a CSV file into a Data Frame with `readtable()`.
⇒ [Listing 8.2](#).

JSON

- Parse a JSON file with `JSON.parse()`.
⇒ [Listing 1.8](#).

HTTP Input

- Create an HTTP request.
⇒ [Listing 1.8](#).
- Convert binary data to a string.
⇒ [Listing 1.8](#).

A.3 Data Structures

Creating Arrays

- Create a range of numbers.
⇒ [Listing 1.2](#).
- Create an array of zero value with `zeros()`.
⇒ [Listing 1.7](#).
- Create an array of one value with `ones()`.
⇒ [Listing 2.4](#).
- Create an array with a repeated value using `fill()`.
⇒ [Listing 7.11](#).
- Create an array of strings.
⇒ [Listing 1.1](#).

- Create an array of numerical values based on a formula.
⇒ [Listing 1.1](#).
- Create an empty array of a given type.
⇒ [Listing 1.3](#).
- Create an array of character ranges.
⇒ [Listing 2.2](#).
- Create an array of tuples.
⇒ [Listing 6.6](#).
- Create an array of arrays.
⇒ [Listing 1.14](#).

Basic Array Operations

- Discover the `length()` of an array.
⇒ [Listing 1.5](#).
- Access elements of an array.
⇒ [Listing 1.5](#).
- Obtain the first and last elements of an array using `first()` and `last()`.
⇒ [Listing 3.32](#).
- Access a sub-array of an array.
⇒ [Listing 1.8](#).
- Apply a function like `sqrt()` onto an array of numbers.
⇒ [Listing 1.1](#).
- Map a function onto an array with `map()`.
⇒ [Listing 8.11](#).
- Append with `push!()` to an array.
⇒ [Listing 1.3](#).
- Convert an object into an array with the `collect()` function.
⇒ [Listing 1.8](#).
- Preallocate an array of a given size.
⇒ [Listing 1.15](#).
- Delete an element from an array or collection with `deleteat!()`.
⇒ [Listing 2.4](#).
- Find the first element of an array matching a pattern with `findfirst()`.
⇒ [Listing 2.4](#).
- Append an array to an existing array with `append!()`.
⇒ [Listing 2.5](#).

- Sum up two equally size arrays element by element.
⇒ [Listing 3.7](#).
- Create a comprehension running over two variables.
⇒ [Listing 3.31](#).
- Stick together several arrays into one array using `vcat()` and
⇒ [Listing 7.10](#).

Further Array Accessories

- Sum up values of an array with `sum()`.
⇒ [Listing 1.6](#).
- Search for a maximal index in an array using `findmax()`.
⇒ [Listing 1.7](#).
- Count the number of occurrence repetitions with the `count()` function.
⇒ [Listing 1.8](#).
- Sort an array using the `sort()` function.
⇒ [Listing 1.8](#).
- Filter an array based on a criterion using the `filter()` function.
⇒ [Listing 1.14](#).
- Find the maximal value in an array using `maximum()`.
⇒ [Listing 2.3](#).
- Count the number of occurrence repetitions with the `counts()` function from `StatsBase`.
⇒ [Listing 2.3](#).
- Reduce a collection to unique elements with `unique()`.
⇒ [Listing 2.5](#).
- Check if a an array is empty with `isempty()`.
⇒ [Listing 3.6](#).
- Find the minimal value in an array using `minimum()`.
⇒ [Listing 3.6](#).
- Accumulate values of an array with `accumulate()`.
⇒ [Listing 3.30](#).
- Sort an array in place using the `sort!()` function.
⇒ [Listing 6.6](#).

Sets

- Check if an element is an element of a set with `in()`.
⇒ [Listing 2.6](#).
- Check if a set is a subset of a set with `issubset()`.
⇒ [Listing 2.6](#).
- Obtain the set difference of two sets with `setdiff()`.
⇒ [Listing 2.5](#).
- Create a set from a range of numbers.
⇒ [Listing 2.6](#).
- Obtain the union of two sets with `union()`.
⇒ [Listing 2.6](#).
- Obtain the intersection of two sets with `intersect()`.
⇒ [Listing 2.6](#).

Matrices

- Obtain the dimensions of a matrix using `size()`.
⇒ [Listing 10.5](#).
- Define a matrix based on a set of values.
⇒ [Listing 1.7](#).
- Define a matrix based on side by side columns.
⇒ [Listing 8.3](#).
- Raise a matrix to a power.
⇒ [Listing 1.7](#).
- Access a given row of a matrix.
⇒ [Listing 1.7](#).
- Stick together two matrices using `vcat()`.
⇒ [Listing 1.7](#).
- Take a matrix and/or vector transpose.
⇒ [Listing 1.7](#).
- Modify the dimensions of a matrix with `reshape()`.
⇒ [Listing 3.13](#).
- us an identity matrix with `I`.
⇒ [Listing 1.7](#).
- Setup a diagonal matrix with `diagm()` and a dictionary.
⇒ [Listing 10.6](#).

- Obtain the diagonal of a matrix with `diag()` .
⇒ [Listing 10.6](#).
- Create a matrix by sticking together column vectors.
⇒ [Listing 1.7](#).

Dictionaries

- Access elements of a dictionary.
⇒ [Listing 1.8](#).
- Create a dictionary.
⇒ [Listing 1.8](#).

Graphs

- Create Graph objects from the package `LightGraphs`.
⇒ [Listing 10.9](#).
- Add edges to Graph objects using `add_edge!()`.
⇒ [Listing 10.9](#).
- Remove edges from Graph objects using `rem_edge!()`.
⇒ [Listing 10.9](#).

Other Data Structures

- Setup a Queue data structure from package `DataStructures`.
⇒ [Listing 10.9](#).
- Insert an element to a Queue data structure using `enqueue!()`.
⇒ [Listing 10.9](#).
- Remove an element from a Queue data structure using `dequeue!()`.
⇒ [Listing 10.9](#).

A.4 Data Frames

Dataframe Basics

- Look at the head of a data frame with `head()`.
⇒ [Listing 4.2](#).
- Get a list of the columns of a data frame and their types with `showcols()`.
⇒ [Listing 4.2](#).Peak at the first few rows of a DataFrame with `head()` lst:_dataframeDetails

- See a summary of the columns of a DataFrame with `showcols()`.
⇒ [Listing 4.2.](#) grp:_dataframeCreation
- Select certain rows of a DataFrame.
⇒ [Listing 4.3.](#)
- Select certain columns of a DataFrame.
⇒ [Listing 4.3.](#)

Dataframe Handling

- Filter all rows of a DataFrame that using a boolean array.
⇒ [Listing 4.3.](#)
- See if data all rows of a DataFrame that using a boolean array.
⇒ [Listing 4.3.](#)
- Check for missing values using `dropmissing()`.
⇒ [Listing 4.3.](#)
- Remove missing values using `dropmissing()`, removing any rows with missing values.
⇒ [Listing 4.4.](#)
- Remove missing values using `skipmissing()` removing specific missing values.
⇒ [Listing 4.4.](#)
- Sort a data frame based on a given column.
⇒ [Listing 8.7.](#)

R Data Sets

- Obtain a data frame from RDataSets with `dataset()`.
⇒ [Listing 8.9.](#)

A.5 Mathematics

Basic Math

- Computer the modulo (remainder) of integer division.
⇒ [Listing 1.15.](#)
- Check if a number is even with `iseven()`.
⇒ [Listing 2.1.](#)
- Take the product of elements of an array using `prod()`.
⇒ [Listing 2.3.](#)

- Round numbers to a desired accuracy with `round()`.
⇒ [Listing 2.8](#).
- Compute the floor of value using `floor()`.
⇒ [Listing 2.10](#).
- Take the product of elements of an array using `*` with `...` as “product”.
⇒ [Listing 5.18](#).
- Represent π using the constant `pi`.
⇒ [Listing 7.16](#).
- Represent Euler’s e using the constant `MathConstants.e`.
⇒ [Listing 7.16](#).

Math Functions

- Compute permutations using the `factorial()` function.
⇒ [Listing 2.3](#).
- Compute the absolute value with `abs()`.
⇒ [Listing 2.3](#).
- Compute the sign function with `sign()`.
⇒ [Listing 8.8](#).
- Create all the permutations of set with `permutations()` from `Combinatorics`.
⇒ [Listing 2.5](#).
- Calculate binomial coefficients with `binomial()`.
⇒ [Listing 2.9](#).
- Use mathematical special functions such as `zeta()`.
⇒ [Listing 2.11](#).
- Calculate the exponential function with `exp()`.
⇒ [Listing 3.6](#).
- Calculate the logarithm function with `exp()`.
⇒ [Listing 3.28](#).
- Calculate trigonometric functions like `cose()`.
⇒ [Listing 3.29](#).
- Create all the combinations of set with `combinations()` from `Combinatorics`.
⇒ [Listing 5.16](#).

Linear Algebra

- Solve a system of equations using the backslash operator.
⇒ [Listing 1.7.](#)
- Use `LinearAlgebra` functions such as `eigvecs()`.
⇒ [Listing 1.7.](#)
- Carry out a Cholesky decomposition of a matrix.
⇒ [Listing 3.32.](#)
- Calculate the inner product of a vector by multiplying the transpose by the vector.
⇒ [Listing 3.33.](#)
- Calculate the inner product by using `dot()`.
⇒ [Listing 8.3.](#)
- Compute a matrix exponential with `exp()`.
⇒ [Listing 10.2.](#)
- Compute the inverse of a matrix with `inv()`.
⇒ [Listing 10.5.](#)
- Compute the Moore-Penrose pseudo-inverse of a matrix with `pinv()`.
⇒ [Listing 8.3.](#)
- Computer the L_p norm of a function with `norm()`.
⇒ [Listing 8.2.](#)
- Computer the QR-factorization of a matrix with `qr()`.
⇒ [Listing 8.3.](#)

Numerical Math

- Find all roots of mathematical function using `find_zeros()`.
⇒ [Listing 1.6.](#)
- Carry out numerical integration using package `QuadGK`.
⇒ [Listing 3.3.](#)
- Carry out numerical differentiation using package `Calculus`.
⇒ [Listing 3.27.](#)
- Carry out numerical integration using package `HCubature`.
⇒ [Listing 3.33.](#)
- Solve a system of equations numerically with `nlsolve()` from package `NLSolve`.
⇒ [Listing 5.7.](#)
- Find a root of mathematical function using `find_zero()`.
⇒ [Listing 5.10.](#)
- Numerically solve a differential equations using the `DifferentialEquations` package.
⇒ [Listing 10.2.](#)

A.6 Randomness, Statistics and Machine Learning

Randomness

- Sample a random number using a prescribed weighting with `sample()`.
⇒ [Listing 1.7](#).
- Get a uniform random number in the range [0, 1].
⇒ [Listing 1.13](#).
- Set the seed of the random number generator.
⇒ [Listing 1.13](#).
- Create a random permutation using `shuffle!()`.
⇒ [Listing 2.8](#).
- Generate a random number from a given range with `rand()`.
⇒ [Listing 2.9](#).
- Generate an array of random uniforms with `rand()`.
⇒ [Listing 2.12](#).
- Generate a random element from a set of values `rand()`.
⇒ [Listing 2.13](#).
- Sample an array of random numbers using a prescribed weighting with `sample()`.
⇒ [Listing 3.8](#).
- Generate multivariate normal random values via `MvNormal()`.
⇒ [Listing 3.34](#).
- Generate an array of standard normal random variables with `randn()`.
⇒ [Listing 4.13](#).
- Generate an array of pseudorandom values from a given distribution.
⇒ [Listing 3.4](#).

Distributions

- Creating a distribution object from the `Distributions` package.
⇒ [Listing 3.4](#).
- Evaluate the PDF (density) of a given distribution.
⇒ [Listing 3.9](#).
- Evaluate the CDF (cumulative probability) of a given distribution.
⇒ [Listing 3.9](#).
- Evaluate the CCDF (one minus cumulative probability) of a given distribution.
⇒ [Listing 5.17](#).

- Evaluate quantiles of a given distribution.
⇒ [Listing 3.9.](#)
- Obtain the parameters of a given distribution.
⇒ [Listing 3.10.](#)
- Evaluate the mean of a given distribution.
⇒ [Listing 3.10.](#)
- Evaluate the median of a given distribution.
⇒ [Listing 3.10.](#)
- Evaluate the variance of a given distribution.
⇒ [Listing 3.10.](#)
- Evaluate the standard deviation of a given distribution.
⇒ [Listing 3.10.](#)
- Evaluate the skewness of a given distribution.
⇒ [Listing 3.10.](#)
- Evaluate the kurtosis of a given distribution.
⇒ [Listing 3.10.](#)
- Evaluate the range of support of a given distribution.
⇒ [Listing 3.10.](#)
- Evaluate the modes (or modes) of a given distribution.
⇒ [Listing 3.10.](#)

Basic Statistics

- Calculate the arithmetic mean of an array.
⇒ [Listing 1.3.](#)
- Calculate a quantile.
⇒ [Listing 1.3.](#)
- Calculate the sample variance of an array.
⇒ [Listing 3.4.](#)
- Calculate the sample standard deviation of an array.
⇒ [Listing 4.6.](#)
- Calculate the median of an array.
⇒ [Listing 4.6.](#)
- Calculate the sample covariance from two arrays.
⇒ [Listing 3.32.](#)
- Calculate the sample correlation from two arrays.
⇒ [Listing 8.3.](#)
- Calculate the sample covariance matrix from a collection of arrays in a matrix.
⇒ [Listing 4.8.](#)

Statistical Inference

- Use the `confint()` function on an hypothesis test.
⇒ **Listing 6.1.**
- Carry out a one sample Z test using `OneSampleZTest()` from the `HypothesisTests` package.
⇒ **Listing 7.1.**
- Carry out a one sample T test using `OneSampleTTest()` from the `HypothesisTests` package.
⇒ **Listing 7.1.**
- Carry out a two sample, equal variance, T test using `EqualVarianceTTest()` from the `HypothesisTests` package.
⇒ **Listing 7.6.**
- Carry out a two sample, non-equal variance, T test using `UnequalVarianceTTest()` from the `HypothesisTests` package.
⇒ **Listing 7.7.**
- Carry out kernel density estimation using `kde()` from package `KernelDensity()`.
⇒ **Listing 4.12.**
- Create and Empirical Cumulative Distribution Function using `ecdf()`.
⇒ **Listing 4.14.**

Linear Models and Generalizations

- Determine a formula for a (generalized) linear model with `@formula`.
⇒ **Listing 8.5.**
- Fit a linear model with `fit()`, `lm()` or `glm()` .
⇒ **Listing 8.5.**
- Calculate the deviance of a linear model with `deviance()` .
⇒ **Listing 8.5.**
- Get the standard error of of a linear model with `stderror()` .
⇒ **Listing 8.5.**
- Get the R^2 value of a linear model with `r2()` .
⇒ **Listing 8.5.**
- Get the fit coefficients of a (generalized) linear model with `coef()` .
⇒ **Listing 8.6.**
- Fit a logistic regression model using package `GLM`.
⇒ **Listing 9.2.**
- Fit a GLMs with different link functions using package `GLM`.
⇒ **Listing 9.3.**

Machine Learning

- Carry out k-means clustering.
⇒ [Listing 9.8](#).
- Carry out principal component analysis.
⇒ [Listing 9.11](#).
- Fit a neural network using package Flux.
⇒ [Listing 9.7](#).

A.7 Graphics

Plotting

- Create two figures, side by side using PyPlot.
⇒ [Listing 1.15](#).
- Plot a mathematical function using PyPlot.
⇒ [Listing 1.9](#).
- Plot two plots on same figure with different colors using PyPlot.
⇒ [Listing 1.9](#).
- Add a legend to a figure using PyPlot.
⇒ [Listing 1.9](#).
- Set the x and y ranges (limits) using PyPlot.
⇒ [Listing 1.9](#).
- Add a title to a figure using PyPlot.
⇒ [Listing 1.9](#).
- Set the aspect ratio of a plot using PyPlot.
⇒ [Listing 1.9](#).
- Set an annotation on a figure using PyPlot.
⇒ [Listing 1.9](#).
- Create a figure with a specified size using PyPlot.
⇒ [Listing 1.14](#).
- Plot individual points not connected by a line using PyPlot.
⇒ [Listing 1.14](#).
- Set the point size of points using PyPlot.
⇒ [Listing 1.14](#).
- Set the points of a plot to be marked by “x” using PyPlot.
⇒ [Listing 8.1](#).

Graphics Primitives

- Draw a line on a figure.
⇒ [Listing 8.2.](#)
- Draw a rectangle on a figure.
⇒ [Listing 8.2.](#)

Statistics Plotting

- Plot a bar graph with `bar()` from PyPlot.
⇒ [Listing 3.1.](#)
- Plot a combination of bars using `plt[:bar]` from PyPlot.
⇒ [Listing 5.13.](#)
- Plot a stem diagram with `stem()` from PyPlot.
⇒ [Listing 2.4.](#)
- Plot a histogram using PyPlot.
⇒ [Listing 1.10.](#)
- Plot a cumulative histogram using PyPlot.
⇒ [Listing 4.13.](#)
- Plot box-plots using `boxplot()` from PyPlot.
⇒ [Listing 7.8.](#)
- Plot a stack plot using `stackplot()` from PyPlot.
⇒ [Listing 4.9.](#)
- Plot a pie chart using `pie()` from PyPlot.
⇒ [Listing 4.10.](#)
- Plot a scatter of points using `scatter()` from PyPlot.
⇒ [Listing 4.11.](#)

Multivariable Function Plotting

- Plot a contour plot using `contourf()` from PyPlot.
⇒ [Listing 3.31.](#)
- Plot a surface plot using `plot_surface()` from PyPlot.
⇒ [Listing 3.31.](#)
- Plot a contour lines using `contour()` from PyPlot.
⇒ [Listing 3.34.](#)

Animation

- Create an animation using PyPlot's `ArtistAnimation`.
⇒ **Listing 1.11**.

Appendix B

Additional Julia Features - DRAFT

Julia has many additional features that we have not touched on in the previous examples. Below is a list of some these. Consult the Julia documentation for more information.

Creation of packages : The nature of our code examples was illustrative, allowing them to run on a standard environment without requiring any special installation. However, once you create code that you wish to reuse, you may want to encapsulate it in a Julia package. This is done via the `generate` command.

Date and time: Julia supports a variety of date and time types and operations. As an example, `DateTime(2019)` constructs a date object that can then be further manipulated. See also the `Dates.jl` package. In many data-science applications, manipulating date and time is commonplace.

Exception Handling: Julia has built-in exception handling support. A key mechanism is the `try`, `throw` and `catch` construct, allowing functions to `throw()` an error if conditions are not met.

Interfaces: Much of Julia's power and extensibility comes from a collection of informal interfaces. By extending a few specific methods to work for a custom type, objects of that type not only receive those functionalities, but they are also able to be used in other methods that are written to generically build upon those behaviors. Iterable objects are particularly useful, and we have used them in several of our examples. In addition, there are methods for indexing, interfacing with Abstract Arrays and Strided Arrays, as well as ways of customizing broadcasting.

Low level TCP/IP Sockets: Julia supports TCP and UDP sockets via the `Sockets.jl` package, which is installed as part of Julia base. The methods will be familiar to those who have used the Unix socket API. For example, `server = listen(ip"127.0.0.1", 2000)` will create a localhost socket listening on port 2000, `connect(ip"127.0.0.1", 2000)` will connect to the socket, and `close(server)` will disconnect the socket.

Metaprogramming: Julia supports "Lisp like" metaprogramming, which makes it possible to create a program that generates some of its own code, and to create true Lisp-style macros which operate at the level of abstract syntax trees. As a brief example, `x = Meta.parse("1 +`

`2"`) parses the argument string into an expression type object and stores it as `x`. This object can be inspected via `drop(x)` (note the `+` symbol, represented by `:+`). The expression can also be evaluated via `eval(x)`, which returns the numerical result of 3.

Modules: Modules in Julia are different workspaces that introduce a new global scope. They are delimited within `module Name ... end`, and they allow for the creation of top-level definitions (i.e. global variables) without worrying about naming conflicts when used together with other code. Within a module, you can control which names from other modules are visible via the `import` keyword, and which names are intended to be public via the `export` keyword.

Parallel processing: Julia supports a variety of parallel computing constructs including green threads, tasks (known as coroutines in Julia) and communication channels between them. A basic macro is `@async` which when used via for example, `@async myFunction()`, would execute `myFunction()` on its own thread.

Rational numbers: Julia supports rational numbers, along with arbitrary precision arithmetic. A rational number such as for example `2/3` is defined in Julia via `2 // 3`. Arithmetic with rational numbers is supported.

Regular expressions: Julia also supports regular expressions, allowing to match strings. For example `occursin(r"^\s*(#)", "# a comment")` checks if `#` appears in the string and returns `true`.

Running external programs: Julia borrows backtick notation for commands from the shell, Perl, and Ruby. However, the behavior of `'Hello world'` varies slightly from typical shell, Perl or Ruby behavior. In particular, the backticks create a `Cmd` object, which can be connected to other commands via pipes. In addition, Julia does not capture the output unless specifically arranged for it. And finally, the command is never run with a shell, but rather Julia parses the syntax directly, appropriately interpolating variables and splitting on words as the shell would, respecting shell quoting syntax. The command is run as Julia's immediate child process, using `fork` and `exec` calls. As a simple example, consider: `run(pipeline('echo world' & 'echo hello', 'sort'))`. This always outputs `'Hello world'` (here both echos are parsed to a single UNIX pipe, and the other end of the pipe is read by the `sort` command).

Strings,: While some of our examples included string manipulation, we haven't delved into the subject deeply. Julia supports a variety of string operations for example, `occursin("world", "Hello, world")` returns `true`.

Unicode and character encoding: Most of the examples in the book were restricted to ASCII characters, however Julia fully supports Unicode. For example `s = "\u2200 x \u2203 y"` yields the string `\forall x \exists y`.

User defined types: In addition to the basic types in the system (e.g. `Float64`), users and developers can create their own types via the `struct` keyword. In our examples, we have not created our own types, however many of the packages define new structs and in some examples of the book, we have referred directly to the fields of these structs. An example is in Listing 8.3 we use `F.Q` to refer to the field `"Q"` in the structure `F`.

Unit testing: As reusable code is developed it may also be helpful to create unit tests for verifying the validity of the code. This allows the code to be retested automatically every time it is

modified or the system is upgraded. For this Julia supports unit testing via the `@test` macro, the `runtests()` function and other objects.

Appendix C

Additional Packages - DRAFT

We have used a variety of packages in this book. These were listed in Section 1.2. However there are many more. Currently, as of the time of writing, there are just over 1,900 registered packages in the Julia ecosystem. Many of these packages deal with numerical mathematics, scientific computing, or deal with some specific engineering or technical application. There are dozens of packages associated with statistics and/or data-science, and we now provide an outline of some of the popular packages in this space that have not been used in our examples.

ARCH.jl is a package that allows for ARCH (Autoregressive Conditional Heteroskedasticity) modeling. ARCH models are a class of models designed to capture a features of financial returns data known as volatility clustering, i.e., the fact that large (in absolute value) returns tend to cluster together, such as during periods of financial turmoil, which then alternate with relatively calmer periods. This package provides efficient routines for simulating, estimating, and testing a variety of ARCH and GARCH models (with GARCH being Generalized ARCH).

AutoGrad.jl is an automatic differentiation package for Julia. It started as a port of the popular Python autograd package and forms the foundation of the Knet Julia deep learning framework. AutoGrad can differentiate regular Julia code that includes loops, conditionals, helper functions, closures etc. by keeping track of the primitive operations and using this execution trace to compute gradients. It uses reverse mode differentiation (a.k.a. back propagation) so it can efficiently handle functions with large array inputs and scalar outputs. It can compute gradients of gradients to handle higher order derivatives.

BayesNets.jl is a package implements Bayesian Networks for Julia through the introduction of the BayesNet type, which contains information on the directed acyclic graph, and a list of conditional probability distributions (CDP's). Several different CDP's are available. It allows to use random sampling, weighted sampling, and Gibbs sampling for assignments. It supports inference methods for discrete Bayesian networks, parameter learning for an entire graph, structure learning, and the calculation of the Bayesian score for a discrete valued BayesNet, based purely on the structure and data. Visualization of network structures is also possible via integration with the *TikzGraphs.jl* package.

Bootstrap.jl is a package for statistical bootstrapping. It has several different resampling methods and also has functionality for confidence intervals.

Convex.jl is a Julia package for Disciplined Convex Programming optimization problems. It can solve linear programs, mixed-integer linear programs, and DCP-compliant convex programs using a variety of solvers, including Mosek, Gurobi, ECOS, SCS, and GLPK, through the MathOptInterface interface. It also supports optimization with complex variables and coefficients.

CPLEX.jl is an unofficial interface to the IBM® ILOG® CPLEX® Optimization Studio. It provides an interface to the low-level C API, as well as an implementation of the solver-independent MathOptInterface.jl. You cannot use *CPLEX.jl* without having purchased and installed a copy of CPLEX Optimization Studio from IBM. This package is available free of charge and in no way replaces or alters any functionality of IBM's CPLEX Optimization Studio product.

CUDAnative.jl is part of the JuliaGPU collection of packages, and provides support for compiling and executing native Julia kernels on CUDA hardware.

Dates.jl is one of Julia's standard libraries and provides the `Date` and `DateTime` types, along with related functions.

DataFramesMeta.jl is a package that provides a series of metaprogramming tools for *DataFrames.jl*, which improve performance and provide a more convenient syntax.

Distances.jl is a package for evaluating distances (metrics) between vectors. It also provides optimized functions to compute column-wise and pairwise distances. This is often substantially faster than a straightforward loop implementation.

FastGaussQuadrature.jl is a Julia package to compute n-point Gauss quadrature nodes and weights to 16 digit accuracy in $O(n)$ time. It includes several different algorithms, including `gausschebyshev()`, `gausslegendre()`, `gaussjacobi()`, `gaussradau()`, `gausslobatto()`, `gausslaguerre()`, and `gausshermite()`.

ForwardDiff.jl is part of the JuliaDiff family, and is a package that implements methods to take derivatives, gradients, Jacobians, Hessians, and higher-order derivatives of native Julia functions (or objects) using forward mode automatic differentiation (AD).

GadFly.jl is a plotting and visualization system written in Julia and largely based on ggplot2 for R. it supports a large number of common plot types and composition techniques, along with interactive features, such as panning and zooming, which are powered by Snap.svg. It renders publication quality graphics in a variety of formats including SVG, PNG, Postscript, and PDF, and has tight integration with *DataFrames.jl*.

GLMNet.jl is a package that acts as a wrapper for Fortran code from glmnet. Also see *Lasso.jl* which is a pure Julia implementation of the glmnet coordinate descent algorithm that often achieves better performance.

Gurobi.jl is a wrapper for the Gurobi solver (through its C interface). Gurobi is a commercial optimization solver for a variety of mathematical programming problems, including linear programming (LP), quadratic programming (QP), quadratically constrained programming (QCP), mixed integer linear programming (MILP), mixed-integer quadratic programming (MIQP), and mixed-integer quadratically constrained programming (MIQCP). It is highly recommend that the Gurobi.jl package is used with higher level packages such as *JuMP.jl* or *MathOptInterface.jl*.

IJulia.jl is a Julia-language back-end combined with the Jupyter interactive environment, which enables interaction with the Julia language using Jupyter/IPython’s powerful graphical notebook on the local machine (rather than using JuliaBox which is server-side).

Images.jl is the main image processing package for Julia. It has a clean architecture and is designed to unify resources from the “machine vision” and “biomedical 3d image processing” communities. It is part of the JuliaImages family of packages.

Interpolations.jl is a package for fast, continuous interpolations of discrete datasets in Julia.

JuliaDB.jl is a package designed for working with large multi-dimensional datasets of any size. Using an efficient binary format, it allows data to be loaded and saved and efficiently, and quickly recalled later. It is versatile, and allows for fast indexing, filtering, and sorting operations, along with performing regressions. It comes with built-in distributed parallelism, and aims to tie together the most useful data manipulation libraries for a comfortable experience.

JuliaDBMeta.jl is a set of macros that aim to simplify data manipulation with *JuliaDB.jl*.

JuMP.jl is a domain-specific modeling language for mathematical optimization embedded in Julia. It supports a number of open-source and commercial solvers (Artelys Knitro, BARON, Bonmin, Cbc, Clp, Couenne, CPLEX, ECOS, FICO Xpress, GLPK, Gurobi, Ipopt, MOSEK, NLOpt, SCS) for a variety of problem classes, including linear programming, (mixed) integer programming, second-order conic programming, semi-definite programming, and non-linear programming (convex and non-convex). JuMP makes it easy to specify and solve optimization problems without expert knowledge, yet at the same time allows experts to implement advanced algorithmic techniques such as exploiting efficient hot-starts in linear programming or using callbacks to interact with branch-and-bound solvers. It is part of the JuliaOpt collection of packages.

Juno.jl is a package that is required to use the Juno development environment. See JunoLab in the organizations section below.

Lasso.jl is a pure Julia implementation of the glmnet coordinate descent algorithm for fitting linear and generalized linear Lasso and Elastic Net models. It also includes: an implementation of the $O(n)$ fused Lasso implementation, an implementation of polynomial trend filtering, and an implementation of Gamma Lasso - a concave regularization path glmnet variant.

Loess.jl is a pure Julia implementation of local polynomial regression (i.e. locally estimated scatterplot smoothing, known as LOESS).

LsqFit.jl is a package providing a small library of basic least-squares fitting in pure Julia. The basic functionality was originally in *Optim.jl*, before being separated. At this time, *LsqFit.jl* only utilizes the Levenberg-Marquardt algorithm for non-linear fitting.

Mamba.jl provides a pure Julia interface to implement and apply Markov chain Monte Carlo (MCMC) methods for Bayesian analysis. It provides a framework for the specification of hierarchical models, allows for block-updating of parameters, with samplers either defined by the user, or available from other packages, and allows for the execution of sampling schemes, and for posterior inference. It is intended to give users access to all levels of the design and implementation of MCMC simulators to particularly aid in the development of new methods. Several software options are available for MCMC sampling of Bayesian models. Individuals

who are primarily interested in data analysis, unconcerned with the details of MCMC, and have models that can be fit in JAGS, Stan, or OpenBUGS are encouraged to use those programs. Mamba is intended for individuals who wish to have access to lower-level MCMC tools, are knowledgeable of MCMC methodologies, and have experience, or wish to gain experience, with their application. The package also provides stand-alone convergence diagnostics and posterior inference tools, which are essential for the analysis of MCMC output regardless of the software used to generate it.

MLBase.jl aims to provide a collection of useful tools to support machine learning programs, including: Data manipulation and preprocessing, Score-based classification, Performance evaluation (e.g. evaluating ROC), Cross validation, and Model tuning (i.e. searching for the best settings of parameters).

MXNet.jl is now part of the Apache MXNet project. It brings flexible and efficient GPU computing and state-of-art deep learning to Julia. Some of its features include efficient tensor/matrix computation across multiple devices, including multiple CPUs, GPUs and distributed server nodes, and flexible symbolic manipulation to composite and construction of state-of-the-art deep learning models.

NLopt.jl provides a Julia interface to the open-source NLopt library for non-linear optimization. NLopt provides a common interface for many different optimization algorithms, including, local and global optimization, algorithms that use function values only (no derivative) and those that exploit user-supplied gradients, as well as algorithms for unconstrained optimization, bound-constrained optimization, and general non-linear inequality/equality constraints. It can be used interchangeably with outer optimization packages such as those from JuMP.

OnlineStats.jl is a package which provides on-line algorithms for statistics, models, and data visualization. On-line algorithms are well suited for streaming data or when data is too large to hold in memory. Observations are processed one at a time and all algorithms use $O(1)$ memory.

Optim.jl is a package that is part of the JuliaNLSolvers family, and provides support for univariate and multivariate optimization through various kinds of optimization functions. Since Optim.jl is written in Julia, it has several advantages: it removes the need for dependencies that other non-Julia solvers may need, reduces the assumptions the user must make, and allows for user controlled choices through Julia's multiple dispatch rather than relying on pre-defined choices made by the package developers. As it is written in Julia, it also has access to the automatic differentiation features via packages in the JuliaDiff family.

Plotly.jl is a Julia interface to the plot.ly plotting library and cloud services, and can be used as one of the plotting backends of the *Plots.jl* package.

Plots.jl is a powerful interface and tool-set for creating plots and visualizations in Julia. It works by connecting commands to various back-ends, which include PyPlot, Plotly, GR and several others. Those familiar with plotting using different back-ends will know that each back-end comes with its own strengths, weaknesses, and syntax. This package aims to simplify the plotting work flow by creating a uniform methodology. It aims to be powerful, intuitive, concise, flexible, consistent, lightweight and smart.

POMDPs.jl is part of the JuliaPOMDP collection of packages and aims to provide an interface for defining, solving, and simulating discrete and continuous, fully and partially observable

Markov decision processes. Note that `POMDP.jl` only contains the interface for communicating MDP and POMDP problem definitions. For a full list of supporting packages and tools to be used along with `POMDPs.jl`, see `JuliaPOMDP`. These additional packages include simulators, policies, several different MDP and POMDP solvers, along with other tools.

`ProgressMetre.jl` is a package that enables the use of a progress meter for long-running Julia operations.

`Reinforce.jl` is an interface for *reinforcement learning*. It is intended to connect modular environments, policies, and solvers with a simple interface. Two packages build on `Reinforce.jl`: `AtariAlgos.jl`, which is an Arcade Learning Environment (ALE) wrapped as `Reinforce.jl` environment, and the `OpenAIGym.jl`, which wraps the open source python library `gym`, released by OpenAI.

`ReinforcementLearning.jl` is a reinforcement learning package. It features many different learning methods and has support for many different learning environments, including a wrapper for the Atari ArcadeLearningEnvironment, and the OpenAI Gym environment, along with others.

`ScikitLearn.jl` implements the popular scikit-learn interface and algorithms in Julia. It supports both models from the Julia ecosystem and those of the scikit-learn library via `PyCall.jl`. Its main features include approximately 150 Julia and Python models accessed through a uniform interface, Pipelines and FeatureUnions, Cross-validation, hyperparameter tuning, and `DataFrames` support.

`SimJulia.jl` is a discrete event process oriented simulation framework written in Julia. It is inspired by the Python SimPy library.

`StatsFuns.jl` is a package that provides a collection of mathematical constants and numerical functions for statistical computing, including various distribution related functions.

`StatsKit.jl` is a convenience meta-package which allows loading of essential packages for statistics in one command. It currently loads the following statistics packages: `Bootstrap`, `CategoricalArrays`, `Clustering`, `CSV`, `DataFrames`, `Distances`, `Distributions`, `GLM`, `HypothesisTests`, `KernelDensity`, `Loess`, `MultivariateStats`, `StatsBase`, and `TimeSeries`.

`StatPlots.jl` is a drop-in replacement for `Plots.jl`. It is slightly less lightweight, but has more functionality and contains many statistical recipes for concepts and types introduced in the `JuliaStats` organization, including `histogram/histogram2d`, `box plot`, `violin`, `marginalhist`, `corrplot/cornerplot`, and `andrewsplot`.

`Tables.jl` combines the best of the `DataStreams.jl` and `Queryverse.jl` packages to provide a set of fast and powerful interface functions for working with various kinds of table-like data structures through predictable access patterns.

`TensorFlow.jl` acts as a wrapper around the popular `TensorFlow` machine learning framework from Google. It enables both input data parsing and post-processing of results to be done quickly via Julia's JIT compilation. It also provides the ability to specify models using native Julia looking code, and through Julia metaprogramming, simplifies graph construction and reduces code repetition.

TensorOperations.jl is a package that enables fast tensor operations using a convenient Einstein index notation.

TimeSeries.jl is a package that provides convenient methods for working with time series data through the introduction of the TimeArray type. It allows for array and column indexing, conditional splitting and mathematical, comparison and logical operations to be performed, along with plotting to be done via the various backends of the *Plots.jl* packages.

XGBoost.jl is a Julia interface of eXtreme Gradient Boosting, or XGBoost. It is an efficient and scalable implementation of gradient boosting framework. It includes efficient linear model solver and tree learning algorithms. The library is parallelized using OpenMP, and it can be more than 10 times faster than some existing gradient boosting packages. It supports various objective functions, including regression, classification and ranking. The package is also made to be extensible, so that users are also allowed to define their own objectives easily. It is part of the Distributed (Deep) Machine Learning Community (dmlc).

Organizations (i.e. collections) of Packages

Much of the Julia package ecosystem on Github is grouped into organizations (or collections) of packages, often based on specific domains of knowledge. Currently there are over 35 different Julia organizations, and some of the more relevant ones for the statistician, data scientist, or machine learning practitioner are listed below.

JuliaCloud is a collection of Julia packages for working with cloud services.

JuliaDiff an informal organization which aims to unify and document packages written in Julia for evaluating derivatives. The technical features of Julia, namely, multiple dispatch, source code via reflection, JIT compilation, and first-class access to expression parsing make implementing and using techniques from automatic differentiation easier than ever before. Packages hosted under the JuliaDiff organization follow the same guidelines as for JuliaOpt; namely, they should be actively maintained, well documented and have a basic testing suite.

JuliaData is a collection of Julia packages for data manipulation, storage, and I/O.

JuliaDiff is an informal organization for solving differential equations in Julia.

JuliaDiffeq is an organization for unifying the packages for solving differential equations in Julia, and includes packages such as DifferentialEquations.jl.

JuliaGeometry is a collection of packages that focus on computational geometry with Julia.

JuliaGPU contains a collection of Julia packages that support GPU computation.

JuliaGraphs is a collection of Julia packages for graph modeling and analysis.

JuliaImages is a collection of packages specifically focused on image processing, and has many useful algorithms. Its main package is Images.jl.

JuliaInterop is a collection of packages that contains many different packages that enable interoperability between Julia and other various languages, such as C++, Matlab, and others.

JuliaMath contains a series of mathematics related packages.

JuliaML contains a series of Julia packages for Machine Learning.

JuliaOpt a collection of optimization-related packages. Its purpose is to facilitate collaboration among developers of a tightly integrated set of packages for mathematical optimization.

JuliaParallel is a collection of packages containing various models for parallel programming in Julia.

JuliaPOMDP is a collection of POMDP packages for Julia.

JuliaPlots is a collection of data visualization plotting packages for Julia.

JuliaPy is a collection of packages that connect Julia and Python.

JuliaStats is the main collection of statistics and Machine Learning packages.

JuliaTeX is a collection of packages for TeX typesetting and rendering in Julia.

JuliaText is a JuliaLang Organization for Natural Language Processing, (textual) Information Retrieval, and Computational Linguistics

Junolab is the landing page for the Juno IDE (integrated desktop environment). Juno is a free environment for the Julia language, is built on the Atom editor, and is a powerful development tool. The `Juno.jl` package defines Juno's frontend API.

Bibliography

- [Ber11] D. P. Bertsekas. *Dynamic programming and optimal control 3rd edition, volume II*, Belmont, MA: Athena Scientific, 2011.
- [CB01] G. Casella and R.L. Berger. *Statistical Inference (2nd ed)*. Cengage Learning, 2001.
- [CLRS09] T.H Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms (3rd ed)*. MIT Press, 2009.
- [MDL13] P de Micheaux, R. Drouilhet, and B. Liquet. *The R Software: Fundamentals of Programming and Statistical Analysis (Statistics and Computing)* . Springer, 2013.
- [DS11] M.H. DeGroot and M.J. Schervish. *Probability and Statistics (4th Edition)*. Pearson, 2011.
- [Dur12] R. Durett. *Essential of Stochastic Processes*. New York: Springer-Verlag, 2012.
- [Fel68] W. Feller. *An Introduction to Probability Theory and Its Applications, Volume 1 (3rd ed)*. New York: Wiley, 1968.
- [HB13] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.
- [Inn18] M. Innes. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 2018.
- [KTB11] D.P. Kroese, T. Taimre, and Z.I. Botev. *Handbook of Monte Carlo Methods*. Wiley, 2011.
- [LG08] A. Leon-Garcia. *Probability, Statistics and Random Processes for Electrical Engineering (3rd ed)*. Pearson, 2008.
- [Litt09] M. L. Littman. *A tutorial on partially observable Markov decision processes* Journal of Mathematical Psychology 2009.
- [Luk42] E. Lukacs. A characterization of the normal distribution. *The Annals of Mathematical Statistics*, 13(1):91–93, 1942.
- [Man07] M. Mandjes. *Large deviations for Gaussian queues: modelling communication networks*. John Wiley & Sons, 2007.
- [Mon17] D.C. Montgomery. *Design and analysis of experiments*. John wiley & sons, 2017.

- [MR13] D.C. Montgomery and G.C. Runger. *Applied Statistics and Probability for Engineers (6th ed)*. Wiley, 2013.
- [Nor97] J.R. Norris. *Markov Chains*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1997.
- [Put14] M. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.
- [SBK75] S. Selvin, M. Bloxham, A. I. Khuri, M. Moore, R. Coleman, G.R. Bryce, J.A. Hagans, T.C. Chalmers, E.A. Maxwell, and G.N. Smith. Letters to the editor. *The American Statistician*, 29(1):67–71, 1975.
- [Van12] S. Van Buuren. *Flexible Imputation of Missing Data*. Chapman and Hall/CRC, 2012.

List of Julia Code

1.1	Hello world and perfect squares	5
1.2	Using a comprehension	6
1.3	Slow code example	9
1.4	Fast code example	9
1.5	Bubble sort	17
1.6	Roots of a polynomial	19
1.7	Steady state of a Markov chain in several ways	21
1.8	Web interface, JSON and string parsing	23
1.9	Basic plotting	24
1.10	Histogram of hailstone sequence lengths	26
1.11	Animated edges of a graph	28
1.12	Working with images	30
1.13	Pseudo random number generation	32
1.14	Estimating π	33
1.15	A linear congruential generator	35
1.16	Random walks and seeds	37
1.17	Using R from Julia	39
1.18	NLP via Python TextBlob	42
2.1	Even sum of two dice	46
2.2	Password matching	48
2.3	The birthday problem	50
2.4	Fishing with and without replacement	52
2.5	Lattice paths	56
2.6	Basic set operations	58
2.7	An innocent mistake with Monte Carlo	59
2.8	Secretary with envelopes	61
2.9	Occupancy problem	63
2.10	Independent events	66
2.11	Defects in manufacturing	68
2.12	Tx Rx Bayes	70
2.13	The Monty Hall problem	72
3.1	A simple random variable	76
3.2	Plotting discrete and continuous distributions	78
3.3	Expectation via numerical integration	79
3.4	Variance of X as a mean of Y	81
3.5	CDF from the Riemann sum of a PDF	84
3.6	The inverse CDF	86

3.7 A sum of two triangular random variables	88
3.8 Sampling from a weight vector	91
3.9 Using the <code>pdf()</code> , <code>cdf()</code> , and <code>quantile()</code> functions with <code>Distributions</code>	92
3.10 Descriptors of <code>Distributions</code> objects	93
3.11 Using <code>rand()</code> with <code>Distributions</code>	94
3.12 Inverse transform sampling	95
3.13 Families of discrete distributions	96
3.14 Discrete uniform dice toss	97
3.15 Coin flipping and the binomial distribution	98
3.16 The geometric distribution	100
3.17 The negative binomial distribution	101
3.18 Comparison of several hypergeometric distributions	103
3.19 The Poisson distribution	105
3.20 Families of continuous distributions	107
3.21 Uniformly distributed angles	107
3.22 Flooring an exponential random variable	109
3.23 Gamma as a sum of exponentials	111
3.24 The gamma and beta special functions	113
3.25 The gamma function at 1/2	114
3.26 Hazard rates and the Weibull distribution	115
3.27 Numerical derivatives of the normal density	116
3.28 Alternative representations of Rayleigh random variables	117
3.29 The Box-Muller transform	118
3.30 The law of large numbers breaks down with very heavy tails	121
3.31 Visualizing a bivariate density	123
3.32 Generating random vectors with desired mean and covariance	126
3.33 Multidimensional integration	128
3.34 Bivariate normal data	128
4.1 Creating a <code>DataFrame</code>	133
4.2 Overview of a <code>DataFrame</code>	133
4.3 Referencing data in a <code>DataFrame</code>	134
4.4 Dealing with missing type entries	135
4.5 Cleaning and imputing data	136
4.6 Summary statistics	137
4.7 Estimating elements of a covariance matrix	139
4.8 Sample covariance	141
4.9 A stack plot	142
4.10 A pie chart	143
4.11 A custom scatterplot	144
4.12 Kernel density estimation	146
4.13 A cumulative histogram plot	148
4.14 Empirical cumulative distribution function	149
4.15 Normal probability plot	150
4.16 Filtering an input file	151
4.17 Searching files in a directory	152
5.1 Distributions of the sample mean and sample variance	154
5.2 Friends of the normal distribution	156
5.3 Are the sample mean and variance independent?	159

5.4	Student's T-distribution	162
5.5	Ratio of variances and the F-distribution	163
5.6	The central limit theorem	166
5.7	A biased estimator	168
5.8	Point estimation via method of moments using a numerical solver	171
5.9	The likelihood function for a gamma distributions parameters	173
5.10	MLE of a gamma distributions parameters	175
5.11	MSE, bias and variance of estimators	177
5.12	A confidence interval for a symmetric triangular distribution	180
5.13	A simple CI in practice	181
5.14	A simple hypothesis test	183
5.15	The distribution of a test statistic under H_0	186
5.16	A randomized hypothesis test	188
5.17	Comparing receiver operating curves	189
5.18	Bayesian inference with a triangular prior	194
5.19	Bayesian inference with a gamma prior	195
5.20	Bayesian inference using MCMC	198
6.1	CI for single sample population with variance assumed known	202
6.2	CI for single sample population with variance assumed unknown	204
6.3	CI for difference in population means with variance known	205
6.4	CI for difference in means with variance unknown and assumed equal	206
6.5	CI for difference in means with variance unknown and not assumed equal	208
6.6	Analyzing the Satterthwaite approximation	209
6.7	Bootstrap confidence interval	211
6.8	Coverage probability for bootstrap confidence intervals	212
6.9	Comparison of sample variance distributions	214
6.10	Actual α vs. α used in variance confidence intervals	216
6.11	Prediction interval given unknown population mean and variance	218
7.1	Inference with single sample when population variance is known	224
7.2	Inference with single sample when population variance unknown	226
7.3	Non-parametric sign test	228
7.4	Comparison of sign test and T-test	229
7.5	Inference on difference of two means (variances known)	232
7.6	Inference on difference of means (variances unknown, assumed equal)	233
7.7	Inference on difference of means (variances unknown, assumed unequal)	235
7.8	Box-plots of data	237
7.9	Sample means for ANOVA	238
7.10	Decomposing the sum of squares	240
7.11	Executing one way ANOVA	242
7.12	Monte Carlo based distributions of the ANOVA F-statistic	244
7.13	Chi-squared test for goodness of fit	249
7.14	Chi-squared for checking independence	251
7.15	Comparisons of distributions of the K-S test statistic	253
7.16	ECDF, actual and postulated CDF's, and their differences	256
7.17	Distributions under different hypotheses	259
7.18	Power curves for different sample sizes	261
7.19	Distribution of the p -value	264
8.1	Polynomial interpolation vs. a line	269

8.2 L1 and L2 norm minimization by MC Simulation	271
8.3 Computing least squares estimates	275
8.4 Using SGD for least squares	277
8.5 Simple linear regression with GLM	281
8.6 The distribution of the regression estimators	283
8.7 Hypothesis tests for simple linear regression	286
8.8 Confidence and prediction bands	287
8.9 The Anscombe quartet datasets	289
8.10 Plotting the residuals and their normal probability plot	292
8.11 Multiple linear regression	294
8.12 Exploring collinearity	296
8.13 Linear regression of a polynomial model	298
8.14 Regression with categorical variables - no interaction effects	300
8.15 Regression with categorical variables - with interaction effects	302
8.16 Simpson's paradox	304
8.17 Basic model selection	307
9.1 Ridge regression with k -fold cross validation	312
9.2 Logistic regression	315
9.3 Exploring generalized linear models	316
9.4 Linear least squares classification	318
9.5 Support vector machines	320
9.6 Random forest	321
9.7 A convolutional neural network	324
9.8 Carrying out k -means via the Clustering package	328
9.9 Manual implementation of k -means	329
9.10 Carrying out hierarchical clustering	330
9.11 Principal component analysis	333
9.12 Principal component analysis on MNIST	335
9.13 Value iteration for an MDP	339
9.14 A Q-Learning example	342
10.1 Trajectory of a predator prey model	347
10.2 Trajectory of a spring and mass system	349
10.3 Two different ways of describing Markov chains	352
10.4 Calculation of a matrix infinite geometric series	357
10.5 Markovian cat and mouse survival	358
10.6 Simulation and analysis using a generator matrix	361
10.7 M/M/1 queue simulation	363
10.8 Discrete event simulation of queues	367
10.9 Discrete event simulation for M/M/1 waiting times	370
10.10 Trajectory of a predator prey model with noise	372
10.11 Kalman filtering	375
10.12 Simple network reliability	378
10.13 Dynamic network reliability	380
10.14 Variance reduction via common random numbers	383
10.15 A case for two RNGs	386

Index

- L_1 norm, 270
- L_2 norm, 270
- α , 185
- k -fold cross validation, 310
- k -means, 327
- p -value, 185, 221
- ?oint process, 104
- DataFrames, 133
- Dataframes, 131
- Dictionary, 25
- FLUX.jl, 322
- GLM package, 267
- QuadGK, 80
- StatsBase, 131
- UnitRange, 28
- floor, 67
- for loop, 6
- if, 17
- missing, 135
- mod, 27
- push
 - , 29
- while, 27
- activation function, 322
- ADAM optimizer, 322
- adaptive control, 341
- adaptive Gauss-Kronrod quadrature, 80
- affine transformation, 125
- agglomerative, 330
- Alternative Hypothesis, 182
- analysis, 153
- analysis of variance, 238
- Analysis of Variance (ANOVA), 221
- analytics, 131
- animation, 27
- ANOVA, 238
- ANOVA table, 242
- Anscombe's quartet, 289
- applied probability, 345, 362
- ARCH.jl, 411
- argument, 11
- array, 6, 17
- array concatenation, 48
- artificial intelligence, 336
- asymptotic approximation, 61
- asymptotically unbiased, 168
- AutoGrad.jl, 411
- autoregressive of order 1 process, 374
- backslash, 273
- bagging algorithm, 321
- balanced design, 237
- bandwidth, 145
- Basel Problem, 68
- Baye's rule, 69
- Bayes estimate, 192
- Bayes' theorem, 69
- Bayesian, 153
- Bayesian inference, 191
- BayesNets.jl, 411
- bell curved, 116
- Bellman equation, 338
- Bellman operator, 338
- Bernoulli trials, 98
- Beta Distribution, 112
- beta distribution, 106, 111
- Beta function, 113
- beta function, 111
- bias, 168
- bias term, 318
- binomial coefficient, 54
- Binomial distribution, 52
- binomial distribution, 96, 98
- birthday problem, 49
- bivariate distribution, 122
- bivariate normal distribution, 128
- block factorial design, 246
- boostrap confidence intervals, 211
- Bootstrap, 211

Bootstrap.jl, 411
 Box-Muller Transform, 118
 Box-plots, 237
 Brownian Bridge, 253
 Brownian Motion, 253
 Bubble Sort, 17

 Calculus.jl, 15
 Cartesian product, 46
 Catalan Number, 54
 categorical variable, 133, 300
 Cauchy distribution, 107, 120
 characteristic function, 83
 checking for independence, 247
 chi-squared, 156
 Chi-squared test, 247
 Chi-squared tests, 221
 Cholesky decomposition, 125
 classification problem, 316
 closed loop, 337
 cloud of points, 268
 clustering, 326
 Clustering.jl, 15
 Code cells, 13
 Collatz conjecture, 26
 collinearity, 295
 combinations, 187
 Combinatorics.jl, 15
 Command Mode, 13
 common random numbers, 37, 382, 383, 385
 compiled language, 8
 complement, 48, 58
 complementary cumulative distribution function, 83
 comprehension, 6, 19
 computational statistics, 192
 conditional density, 123
 conditional expectation, 267
 conditional probability, 67
 conditional statement, 17
 confidence bands, 287
 confidence interval, 180
 Confidence intervals, 153
 confidence level, 180
 confusion matrix, 318
 conjugacy, 192
 conjugate prior, 192, 195
 consistency, 169

 Constructor, 91
 contingency table, 250
 continuous distributions, 106
 continuous random variable, 77
 continuous time Markov chain, 360
 continuous uniform distribution, 106, 107
 contraction, 339
 control policy, 337
 control theory, 373
 Convex.jl, 412
 convolution, 90
 correlation coefficient, 124
 covariance, 124
 covariance matrix, 124
 coverage probability, 212
 CPLEX.jl, 412
 credibility intervals, 192
 critical values, 226
 cross validation, 317
 CSV.jl, 15
 CUDA native.jl, 412
 cumulative distribution function, 83
 cumulative histogram, 148

 dash-boarding, 131
 data, 153, 316
 data cleaning, 131, 132
 data cleansing, 131
 data configurations, 131
 data scraping, 131
 data visualization, 332
 DataFrames, 132
 DataFrames.jl, 15
 DataFramesMeta.jl, 412
 DataStructures.jl, 15
 Dates.jl, 412
 De Morgan's laws, 61
 decision trees, 321
 Decision trees and random forest, 317
 DecisionTree.jl, 15
 decomposition of the sum of squares, 239
 Decreasing Failure Rate (DFR), 115
 deep neural networks, 277, 317, 321
 degrees of freedom, 162, 241, 279
 denominator degrees of freedom, 163
 dependent variable, 267
 descriptive statistics, 131
 design, 279

design matrix, 273
development speed, 1
diagonalize, 332
difference, 57
difference equation, 346
`DifferentialEquations.jl`, 15
diffusion process, 253
digamma function, 175
dimension reduction, 326
discount factor, 337
discrete distributions, 96
discrete event simulation, 365
discrete random variable, 77
discrete time, 20
discrete uniform distribution, 96, 97
disjoint, 59
dispersion, 80
`Distances.jl`, 412
distributed as, 156
distribution without a mean, 121
`Distributions.jl`, 15
divisive, 330
Doob-Gillespie algorithm, 361
dynamic programming principle, 338
dynamical systems, 346

edges, 27, 376
Edit Mode, 13
elastic net, 310
elements, 57
embedded Markov chain, 361
empirical distribution function, 148
epoch, 326
equality operator, 27
equilibrium point, 346
Erlang, 369
error function, 322
errors, 270
estimation error, 373
estimator, 167
Euclidean norm, 270
event, 365
event schedule, 366
events, 45
excess kurtosis, 83
expected value, 79
experiment, 45
experimental design, 222, 246, 261

experimental studies, 131
exploration and exploitation, 341
exponential distribution, 106, 108
exponential family, 312

F distribution, 239
F-distribution, 163
F-test, 239
F-value, 239
family of probability distributions, 96
`FastGaussQuadrature.jl`, 412
features, 316
feedback control, 337
finite horizon, 337
first quartile, 86, 137
first step analysis, 55
floor function, 109
`Flux.jl`, 15
folded normal distribution, 197
`ForwardDiff.jl`, 412
Fourier transform, 83
frequentist, 153
full rank, 273
fully convoluted layer, 323
function, 18
functional programming, 4

`GadFly.jl`, 412
gamma distribution, 106, 110
gamma Function, 110
Gaussian (normal) distribution, 106
Gaussian distribution, 116
Generalized Linear Models, 280
Generalized Linear Models (GLM), 312
generator matrix, 360
geometric distribution, 96, 99
Gibbs Sampling, 197
Gillespie algorithm, 361
Glivenko Cantelli Theorem, 253
`GLM.jl`, 15
`GLMNet.jl`, 412
global variable, 38
goodness of fit, 247
gradient descent, 276
Grahm matrix, 273
graph, 27
graphics primitives, 27
graphs, 376

- Gurobi.jl, 412
- hailstone sequence, 26
 Hamiltonian Monte Carlo, 197
 hazard function, 83
 hazard rate function, 114
 HCubature.jl, 15
 hierarchical clustering, 327, 330
 high dimensional, 310
 high dimensional data, 316
 histogram, 26
 hitting time, 356
 HTTP.jl, 15
 hyper parameter, 309
 Hypergeometric distribution, 52
 hypergeometric distribution, 96, 102
 hyperparameters, 191, 195, 322
 Hypothesis Testing, 182
 Hypothesis tests, 153
 HypothesisTests.jl, 15
- IJulia.jl, 413
 Images.jl, 413
 imaginary plane, 28
 imputation, 136
 inclusion exclusion formula, 59
 Increasing Failure Rate (IFR), 114
 independence, 66
 independent, 122
 independent and identically distributed (i.i.d.),
 154
 independent events, 66
 independent variables, 267
 indicator function, 84, 173
 infinite horizon average reward, 337
 infinite horizon expected discounted reward, 337
 infinite matrix geometric series, 357
 inflection points, 116
 initial state, 346
 integral transform, 87
 intensity, 105, 360
 inter quartile range (IQR), 137
 inter-quartile range, 86
 interaction effect, 300, 302
 Interpolations.jl, 413
 interpreted language, 8
 intersection, 57, 59, 66
 inverse CDF, 85
 inverse cumulative distribution function, 83
 inverse function, 85
 inverse link function, 312
 inverse probability transform, 94
 inverse transform sampling, 94
 irreducible, 358
- Java Script Object Notation, 22
 joint probability density function, 122
 JSON.jl, 15
 Julia Box, 4
 Julia Computing, 4
 Julia language and framework, 2
 Julia methods, 8
 JuliaBox, 11, 12
 JuliaCloud, 416
 JuliaData, 416
 JuliaDB.jl, 413
 JuliaDBMeta.jl, 413
 JuliaDiff, 416
 JuliaDiffEq, 416
 JuliaGeometry, 416
 JuliaGPU, 416
 JuliaGraphs, 416
 JuliaImages, 416
 JuliaInterop, 416
 JuliaMath, 417
 JuliaML, 417
 JuliaOpt, 417
 JuliaParallel, 417
 JuliaPlots, 417
 JuliaPOMDP, 417
 JuliaPy, 417
 JuliaStats, 417
 JuliaTeX, 417
 JuliaText, 417
 jump chain, 361
 JuMP.jl, 413
 Juno.jl, 413
 Junolab, 417
 Jupyter, 4
 Jupyter notebooks, 12
 just-in-time (JIT), 4
- k-means clustering, 328
 Kalman filtering, 373
 KDE, 145
 kernel, 11

- kernel density estimation, 145
kernel function, 145
KernelDensity.jl, 16
Kolmogorov distribution, 253
Kolmogorov-Smirnov statistic, 248
Kolmogorov-Smirnov test, 247
Kolmogorov-Smirnov tests, 221
kurtosis, 83

labelled data, 316
labels, 316
lack of memory, 108
Laplace transform, 83, 88
LASSO, 310
Lasso.jl, 413
latent variable, 295
lattice path, 54
law of total probability, 68
learning rate, 276, 326
learning speed, 2
least absolute shrinkage and selection operator, 310
least squares, 267, 272
least squares estimators, 274
least squares normal equations, 274
least squares problem, 273
Leontief series, 357
less than full rank, 295
levels, 237, 300
LIBSVM.jl, 16
LightGraphs.jl, 16
likelihood, 172
limiting distribution, 358
linear congruential generators, 35
Linear least squares classifiers, 317
Linear Minimum Mean Square Error (LMMSE), 374
linear programming, 338
linear regression, 279
linear regression with one variable, 279
linear relationship, 124
linear system theory, 373
linear systems with additive noise, 373
LinearAlgebra.jl, 16
link function, 312
LLVM, 4
Loess.jl, 413
log-likelihood function, 174

logistic distribution, 214
logistic function, 85, 313
logistic model, 313
logistic regression, 313
logit function, 313
longitudinal studies, 131
Lorentz distribution, 120
loss function, 322
Lotka-Volterra equations, 346
LsqFit.jl, 413
Luenberger observer, 373

machine learning, 316
macro, 9
Mamba.jl, 413
marginal densities, 122
marginal distribution, 251
Markdown, 13
Markdown cells, 13
Markov chain, 19
Markov Chain Monte Carlo, 197
Markov chains, 350
Markov Decision Processes, 336
Markov jump process, 360
Markov property, 354
matrix exponential, 349
maximum, 137
Maximum likelihood estimation, 172
maximum likelihood estimation, 167
maximum likelihood estimator, 172
mean, 79
Mean Squared Error, 168
mean vector, 124
median, 86, 137
memoryless property, 354
Mersenne Twister, 36
meta-programming, 4
method, 8, 11
method of moments, 167, 170
Metropolis Hastings, 197
minimum, 137
Minimum Mean Square Error (MMSE), 374
mixed discrete and continuous distribution, 86
mixture model, 146
MLBase.jl, 414
MNIST, 323
MNIST digits dataset, 316
model, 153

model assumptions, 289
 model selection, 306
 modulus, 67
 moment generating function, 83
 moments, 79
 Monte Carlo Markov Chains, 197
 Monte Carlo simulation method, 31
 Monty Hall problem, 71
 Moore-Penrose pseudo-inverse, 273
 multi-variate distribution, 122
 multi-variate normal distribution, 127
 multicollinearity, 295
 multiple dispatch, 4, 8, 11
 MultivariateStats.jl, 16
 MXNet.jl, 414

 Natural Language Processing, (NLP), 41
 negative binomial distribution, 96, 101
 nested loops, 17
 network input and output, 4
 network reliability, 376
 neural network, 321
 Neural networks, 317
 neural networks architecture, 322
 neurons, 322
 NLopt.jl, 414
 NLsolve.jl, 16
 nominal variable, 133
 non-central T-distribution, 261
 non-parametric, 222
 non-parametric test, 187, 227
 non-singular, 273
 norm, 339
 normal distribution, 116
 normal equations, 274
 normality assumption, 156
 not-equals comparison operator, 27
 Null Hypothesis, 182
 number theory, 35
 numerator degrees of freedom, 163
 numerical computations, 4
 numerical variable, 133

 object oriented programming, 4
 objective function, 322
 observability, 374
 observational studies, 131
 Observations in tuples, 131

 offered load, 363
 one cold encoding, 323
 one sided hypothesis test, 221
 one-way anova, 238
 one-way ANOVA test, 239
 OnlineStats.jl, 414
 open loop, 337
 Optim.jl, 414
 optimization function, 322
 optimization objective, 336
 optional typing, 4
 order statistics, 137
 ordinal variable, 133, 300
 Ordinary Differential Equation, 346
 outer product, 124
 over fitting, 268
 overall mean, 236
 overfitting, 317

 package manager, 11
 package manager mode, 14
 parallel computing, 5
 parameter, 96, 106
 parameter space, 96
 Pareto optimal frontier, 1
 Partially Observable Markov Decision Processes, 336
 partition, 68
 passed by reference, 18
 PDF, 77
 Pearson's chi-squared test, 249
 percentile, 86
 periodic, 356
 Perron Frobenius Theorem, 20
 Plotly.jl, 414
 Plots.jl, 414
 PMF, 75
 Point estimation, 153
 point estimation, 167
 Poisson distribution, 105
 poisson distribution, 96
 Poisson process, 104
 polar coordinate, 118
 policy iteration, 338
 polynomial interpolation, 268
 POMDPs.jl, 414
 pooled sample variance, 206
 population, 153

positive recurrent, 358
posterior distribution, 153, 191
posterior mean, 192
posterior outcome, 69
Power, 183
power, 258
power curve, 261
predator prey model, 346
prediction bands, 287
prediction interval, 218
prediction intervals, 201
predictor, 267
Principal Component Analysis (PCA), 332
principal component analysis (PCA), 327
prior condition, 69
prior distribution, 153, 191
probability, 45
Probability Density Function, 77
probability distribution, 75
probability function, 45
probability generating function, 83
Probability Mass Function, 75
probability measure, 45
probability model, 54
probability space, 45, 46
probability transition law, 351
probability vector, 91
procedural programming, 4
process oriented, 366
ProgressMetre.jl, 415
proposal density, 197
pseudorandom number generation, 31
PyCall.jl, 16
PyPlot.jl, 16

Q-function, 338
Q-learning, 341
Q-learning update equation, 341
Q-table, 343
QR factorization, 273
QuadGK.jl, 16
quantile, 86, 137
quartiles, 86
queueing theory, 86, 362

R squared, 279
random experiment, 45, 46
random forest, 321

random sample, 154
random variables, 75
random vector, 122
Random.jl, 16
randomization test, 187
rank one matrices, 332
Rayleigh Distribution, 117
Rayleigh distribution, 106
Rayleigh fading, 117
RCall.jl, 16
RDataSets.jl, 16
Read Evaluate Print Loop (REPL), 12
Receiver Operating Curve, 189
recurrence relation, 55
recurrent, 355
regenerative simulation, 358
registered, 14
regression, 267
regression analysis, 267
regression problem, 316
regularization, 309
regularization parameter, 309
Reinforce.jl, 415
Reinforcement Learning, 336
reinforcement learning, 345, 415
ReinforcementLearning.jl, 415
rejection region, 185, 222
reliability of the network, 377
reliability theory, 345
REPL command line interface, 11
residual plot, 291
residuals, 270, 291
reward function, 336
Riccati equation, 374
ridge regression, 310
Riemann sum, 84, 368
Riemann Zeta Function, 68
robotics, 336
Roots.jl, 16
runtime speed, 1

sample correlation, 139
sample covariance, 139
sample covariance matrix, 139, 140
sample mean, 137, 154
sample space, 45
sample standard deviation, 137
sample variance, 137, 154

sampling without replacement, 103
 Satterthwaite approximation, 207, 235
 scale parameter, 110
 scheduling the event, 366
 scientific computing, 1
 scientific programming language, 4
 ScikitLearn.jl, 415
 scree plot, 332, 334
 seed, 32
 sensors, 373
 sets, 57
 shape parameter, 110
 shrinkage estimator, 310
 sigmoid function, 85, 313
 sign test, 227
 Silverman's rule, 145
 SimJulia.jl, 415
 Simple Hypothesis Test, 183
 simple linear regression, 279
 Simpson's paradox, 304
 Single sample, 131
 singular value decomposition, 273
 skeleton, 361
 skewed to the left, 83
 skewed to the right, 83
 skewness, 82
 softmax function, 323
 sorted sample, 137
 special function, 113
 SpecialFunctions.jl, 16
 splat operator, 63
 squared coefficient of variation, 111
 stack plot, 142
 standard error, 137
 standard multi-variate, 128
 standard normal, 116
 state, 345, 346
 state space, 351
 stationary deterministic Markov policies, 337
 stationary distribution, 20, 358
 statistic, 154
 statistical computing, 1
 statistical inference, 153
 statistical tables, 223
 statistics, 154
 Statistics.jl, 16
 StatPlots.jl, 415
 StatsBase.jl, 16
 StatsFuns.jl, 415
 StatsKit.jl, 415
 stepwise regression, 306
 stochastic approximation, 342
 stochastic control, 345
 stochastic differential equations, 371
 stochastic dynamic programming, 336
 stochastic gradient descent, 277, 322
 stochastic operations research, 345, 362
 stochastic optimal control, 336
 stochastic process, 104, 253
 stochastic recursive sequence, 351
 strongly typed, 4
 Student T-distribution, 161
 student T-distribution, 156
 subset, 57
 subtype, 11
 sufficient statistics, 175
 sum of squares error, 240
 sum of squares total, 239
 sum of squares Treatment, 240
 supertype, 11
 supervised learning, 309
 Support Vector Machines (SVM), 317, 319
 survey sampling, 131
 symmetric probability function, 46
 T-statistic, 161, 225
 T-test, 225, 226
 Tables.jl, 415
 TensorFlow, 322, 415
 TensorFlow.jl, 415
 TensorOperations.jl, 416
 test set, 317
 test statistic, 185, 221
 the birthday paradox, 49
 the Central Limit Theorem, 165
 theory of non-negative matrices, 356
 third quartile, 86, 137
 Tikhonov regularization, 310
 time homogenous, 354
 time series, 131
 TimeSeries.jl, 416
 trained, 322
 training set, 317
 trajectory, 346
 transformation of variables, 298
 transient, 355

Transition Probability Matrix, 20
transition probability matrix, 351
tuples, 46
two language problem, 2
Two samples, 131
two sided hypothesis test, 221
two-way ANOVA, 246
type, 11
Type I Error, 183
Type II Error, 183
type inference, 4, 11
type instability, 11

unbiased, 168
unbiased estimator, 154
uncertainty bands, 287
uncorrelated, 124
Unicode, 4
union, 57
unit circle, 27
univariate, 127
universal set, 58
unlabelled data, 326
unregistered, 14
unsupervised learning, 309, 326
user defined types, 11

validation set, 317
value function, 337
value iteration, 338
Vandermonde matrix, 268
variable transformations, 294
variance, 79, 80
variance reduction, 382
Venn diagram, 61
vertices, 27, 376

Wald-Wolfowitz runs test, 291
warm up sequence, 199
Weibull Distribution, 114
Weibull distribution, 106, 114
weight decay, 326
with replacement, 52
without replacement, 51

XGBoost.jl, 416

Yule-Simpson effect, 304

z transform, 83