Introduction to R Programming for "Data, Probability and Statistics"

By

Jiajing Sun

School of Mathematics, University of Birmingham
Ring Rd N, Birmingham B15 2TS
Email: j.sun.5@bham.ac.uk

Contents

Preface

R is a free and open-source programming language, developed by Ross Ihaka and Robert Gentleman at the University of Auckland, and released in 1995. It can be used for data analysis, statistical modeling, visualization, machine learning, and more. R is recognized for its flexibility, user-friendly syntax, and cross-platform compatibility, functioning seamlessly on Windows, Mac, and Linux. With a comprehensive library system encompassing packages like `lme4`, `glmnet`, `ggplot2`, and `randomForest`, R exhibits versatility in processing a variety of data formats and performing a wide range of tasks, from data cleaning to intricate statistical modeling. Its vibrant community consistently contributes new packages and tools, amplifying its capabilities. When contrasted with matrix languages like Gauss or MATLAB, R's open-source foundation, inherent statistical tools, amicable user community, and compatibility with other languages such as Python and Java place it as the foremost choice for data professionals.

The "Introduction to R Programming for Data, Probability, and Statistics" emerges as a revitalized version of the data science facet of a previous module named "Probability and Statistics". This foundational module provided insights into the world of probability theory, a mathematical framework to understand randomness and uncertainty, and found applications in domains as varied as genetics, finance, and artificial intelligence. Topics like axiomatic probability, models such as the binomial, Poisson, and normal distributions, properties of random variables, mathematical inequalities, and key statistical estimation techniques were central to this module. Recognizing the seismic shifts in industry needs and the increasing emphasis on data-driven decision-making, the School of Mathematics, University of Birmingham, during its 2023 curriculum review, chose to weave in the data science component into this module. The choice of R, a globally recognized powerhouse in statistical computing, ensures students are equipped with the hands-on expertise that the contemporary research landscape and industrial scenarios demand.

CHAPTER 1

History of the R Language

R is a programming language used for statistical computing and graphics, created by Ross Ihaka and Robert Gentleman in 1993. Originally, R was developed as a free alternative to the S language. The S language, developed at Bell Laboratories, is a statistical language used for data analysis and plotting, but it requires a licensing fee. It can be said that R is a dialect of S.[1] Therefore, to understand the history of R, one must first understand the S language.

1. The S Language

The S language was co-developed by John Chambers, along with his colleagues at Bell Laboratories, Rick Becker and Allan Wilks. In 1976, the S language was launched as an internal statistical analysis environment as part of Fortran libraries.[2] Early versions of the S language didn't even include functions for statistical modeling.

In 1980, the first version of S began to be distributed outside of Bell Laboratories, and by 1981, the source code version was also made available. In 1984, the research team at Bell Labs published two books: S: An Interactive Environment for Data Analysis and Graphics [1] and Extending the S System [2]. That same year, the S source code began to be sold for educational and commercial purposes exclusively through AT&T Software.

In 1988, the system was rewritten in C, and the S language began to evolve to become more like the system (Version 3) we use today. The fourth version of the S language was released in 1998, which is the version in use today. For statistical analysis features of the S software, you can refer to the book Statistical Models in S written by John Chambers and others [4], as well as Programming with data: A guide to the S language by John Chambers [3]. Since 1998, the foundations of the S language haven't undergone significant changes. In 1998, the S language won the very prestigious ACM Software System Award.[3]

Since the early 90s, the lifecycle of the S language has had its twists and turns. In 1993, Bell Labs granted exclusive rights to develop and sell the S language to StatSci (later known as Insightful Corp.). In 2004, Insightful purchased the rights to the S language for 2 million USD. Insightful's product was named S-PLUS, which, as the name implies, was implemented using the S language. Because Insightful built numerous graphical user interfaces (GUIs) on top of it, it

[1]For details, see: `https://stat.ethz.ch/~www/SandR.html`.

[2]The first versions of SPSS and SAS Analytics were also composed of subroutines that could be called from a program (Fortran or others).

[3]ACM refers to the Association for Computing Machinery, one of the most influential organizations in the field of computer science. The ACM Software System Award began in 1995, an annual award by ACM, recognizing individuals or teams who have made outstanding contributions in the software field.

became known as "S-PLUS".[4] In 2008, TIBCO purchased Insightful for 25 million USD. To date, TIBCO owns the rights to the S language and is its exclusive developer.

2. The R Language

The R language was initially developed by Ross Ihaka and Robert Gentleman at the Department of Statistics, University of Auckland. Its design was primarily for a programming language that can efficiently perform data analysis and visualization. The name "R" was derived from the first letter of the founders' names. Since its inception, the R language has evolved into a powerful tool for data science, finding extensive applications in various research and practical domains.

The origins of R trace back to the early 1990s when Professor Ihaka was designing user-friendly statistical software for his students and colleagues. They required an open-source, cross-platform software to ensure statistical analyses were not dependent on commercial software. Initially, Professor Ihaka adopted the S language as a foundation, modifying and expanding it to form the prototype of the R language. Statistician Martin Mächler convinced Ihaka and Gentleman to release R as open-source software under the GNU General Public License,[5] and in 1995 they released the first version of R and began teaching R language courses at the University of Auckland[9]. R quickly became mainstream for statistical analysis.

The R Core Team was formed in 1997, responsible for managing and maintaining the development of the R language[6]. As of January 2022, it consists of Chambers, Gentleman, Ihaka, and Mächler, as well as statisticians Douglas Bates, Peter Dalgaard, Kurt Hornik, Michael Lawrence, Friedrich Leisch, Uwe Ligges, Thomas Lumley, Sebastian Meyer, Paul Murrell, Martyn Plummer, Brian Ripley, Deepayan Sarkar, Duncan Temple Lang, Luke Tierney, and Simon Urbanek, and computer scientist Tomas Kalibera. Stefano Iacus, Guido Masarotto, Heiner Schwarte, Seth Falcon, Martin Morgan, and Duncan Murdoch were former members. The Comprehensive R Archive Network (CRAN) was created by Kurt Hornik and Fritz Leisch in 1997 to host R's source code, executables, documentation, and user-created packages[8]. It was named and patterned after the Comprehensive TeX Archive Network and the Comprehensive Perl Archive Network. Initially, CRAN had three mirrors and 12 contributed packages. By December 2022, it hosted 103 mirrors and 18,976 contributed packages.

In 2003, the R language formally released its version. In April 2003, the R Foundation, a nonprofit organization, was officially established, providing further support for the R project. In 2004, R was licensed as open-source under the GNU General Public License. The download link for R

[4]S-PLUS offers a variety of graphical user interfaces (GUIs), helping users quickly create analyses, making analysis results more comprehensible and communicable. This includes a data editor for importing, editing, and managing datasets; plotting interfaces for generating various statistical graphics, such as histograms, scatter plots, line plots, etc.; model fitting interfaces for fitting various statistical models, such as linear regression, logistic regression, ANOVA, etc.; data exploration interfaces for exploring dataset features and patterns, like clustering analysis, principal component analysis, etc.; and data visualization interfaces for generating interactive data visualizations, like interactive maps, time series plots, etc.

[5]The GNU General Public License (GNU GPL or GPL) is a series of widely used free software licenses that guarantee end users the freedom to run, study, share, and modify the software.

is: `https://cran.r-project.org/`. R is available for Linux (Debian, Fedora/Redhat, Ubuntu), macOS, and Windows, and readers can choose based on their operating system.

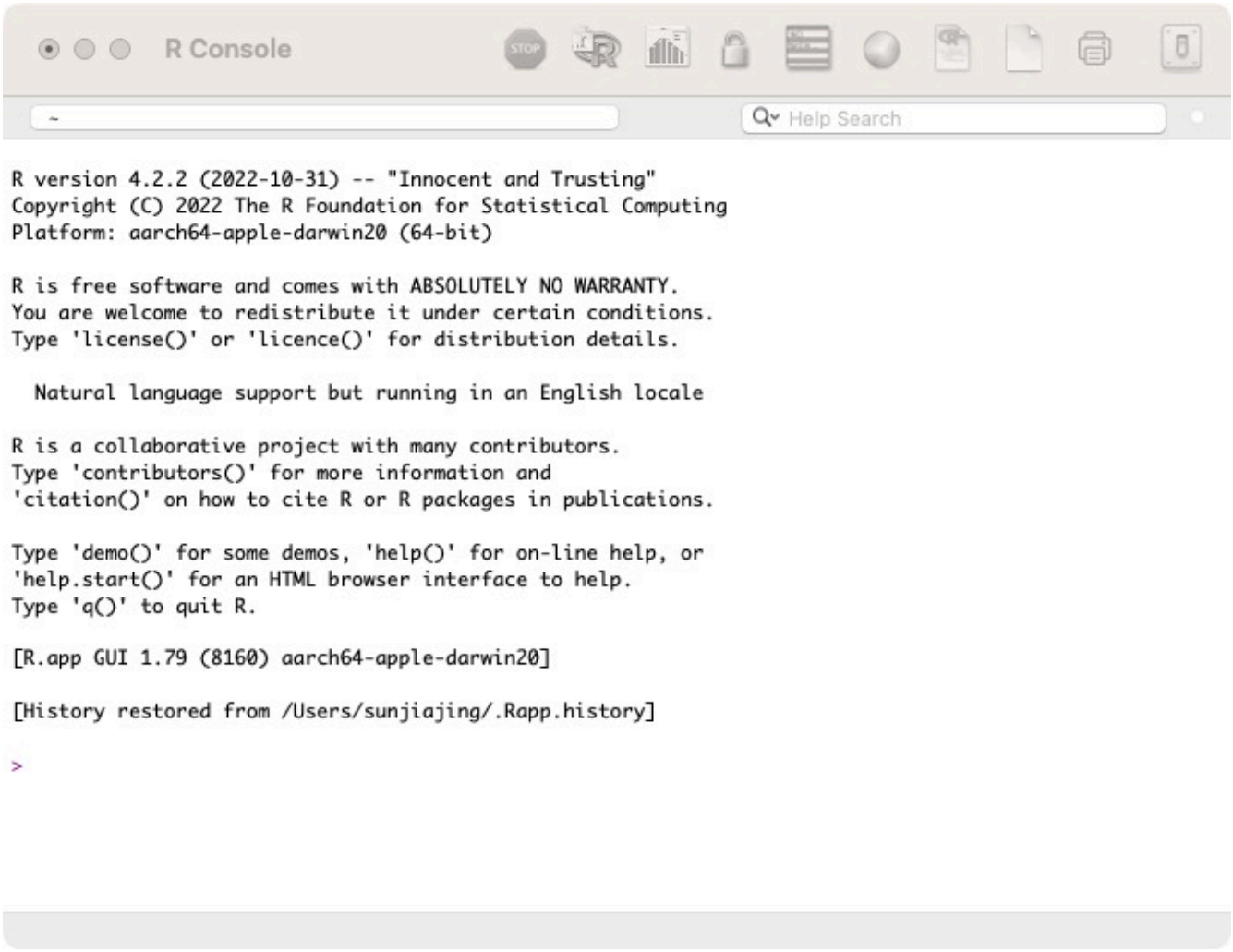


FIGURE 1. R software interface (Version 4.2.2)

3. R's Integrated Development Environment - RStudio

If you prefer integrated development environments (IDE), consider installing RStudio. RStudio is an IDE specifically developed for the R language by Posit, PBC (Public-benefit corporation).[6] RStudio provides a convenient, efficient, and extensible development environment for R developers, greatly enhancing development efficiency and code quality. If you are new to R or prefer the GUI version of R, then you'll appreciate RStudio. Like other programming languages, R is extended or developed via user-written functions. To promote R programming, Posit designed this integrated development environment (IDE), known as RStudio. Both RStudio Desktop and RStudio Server offer free and commercial versions.

[6]The original name of Posit was RStudio, which was renamed to Posit in 2022. See: `https://posit.co/about/`.

The chief scientist and founder of RStudio is Hadley Wickham, one of the most respected data scientists in the R community and an author and contributor to many R packages. Before creating RStudio, his primary work was in R package development and data science research. The first version of RStudio was released in 2011 as a free open-source project. Since then, RStudio has evolved into a powerful and popular R development environment. The owner of RStudio is Posit PBC, a company founded by J.J. Allaire. J.J. Allaire was the developer of the ColdFusion language. He co-created RStudio with Hadley Wickham. In addition to the RStudio IDE, Posit PBC has developed other R-related products, such as Shiny, an R package for creating interactive web applications, and R Markdown, a documentation format that combines R code with Markdown.

What is an IDE? An IDE (Integrated Development Environment) is a software suite that combines the basic tools required for software development. Unlike many other statistical software packages that use a graphical user interface, R users mainly interact through the command line. Thus, an IDE for R must allow commands to be issued interactively. In this regard, R is not unique; other interactive scientific programming languages also have mature IDEs. Besides a console for issuing commands, IDEs also include a source code editor, object browser, object editor, integration with base documentation, plotting management tools, etc. The source code editor, vital for programming, features rich shortcuts, automatic source code formatting, bracket assistance, keyword highlighting, code folding, navigation between files, interfaces for compiling and running software, etc. Object browsers and editors allow users to quickly identify the type and value of each defined variable and to inspect or modify objects.

RStudio has a user-friendly interface, rich features, and an intuitive interface. It mainly consists of four panes:

(1) **Source Editor** helps you open, edit, and run programs and is located in the top left panel of the screen.
(2) **Console** is where you can input and execute code immediately, also known as the command window. It's located in the bottom left corner of the screen.
(3) **Environment Pane** displays the objects (data frames, arrays, values, functions) you have in your environment or workspace.
(4) The last pane, located in the bottom right of the screen, contains multiple tabs.
    (a) **Plots Tab** displays the graphics you create.
    (b) **Connections Tab** helps connect to existing data sources, functioning similarly to data import, assisting in crafting an R statement for data processing in R.
    (c) **Packages Tab** shows installed and installable packages.
    (d) **Help Tab** allows you to search R documentation for assistance. It's also where help appears when requested from the console.

If you wish to rearrange the panes, you can navigate to Tools in the menu, click on Global Options, find Pane Layout, and customize as needed.

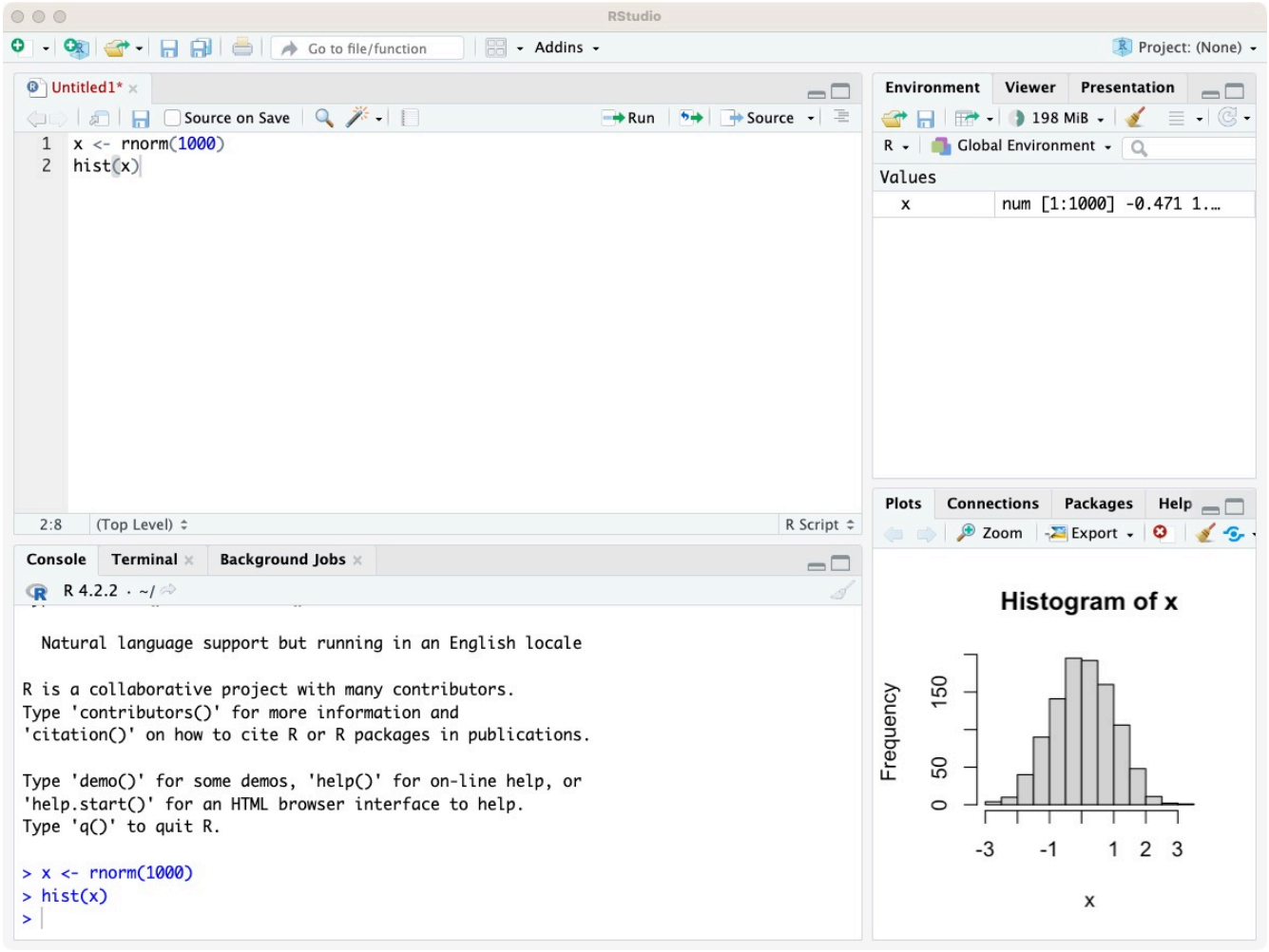


FIGURE 2. User interface of RStudio

RStudio supports multiple operating systems, and the OS support depends on the version of the IDE. RStudio Desktop is available for Windows, macOS, and Linux. RStudio Server can run on Debian, Ubuntu, Red Hat Linux, CentOS, openSUSE, and SLES. Users can manage code versions directly in RStudio, performing operations like code comparison, merging, etc. RStudio also integrates an R package manager, allowing users to search, install, and update R packages conveniently.

Moreover, RStudio supports various plotting and report generation capabilities like ggplot2, Shiny, knitr, and rmarkdown. ggplot2 is a potent plotting tool that can render various statistical charts, such as scatter plots, histograms, and box plots. Additionally, the R language has a series of data visualization packages like lattice, leaflet, playwith, ggvis, and ggmaps. R also provides partial map plotting capabilities, and readers interested in spatial data analysis can refer to `https://rspatial.org/`. Shiny is an R package that allows for the easy building of interactive web applications from R, hosting standalone applications on the web, embedding them into R Markdown documents, or building dashboards (themes). Additionally, you can extend Shiny

applications using CSS themes, htmlwidgets, and JavaScript actions.[7] Knitr and rmarkdown are two report generation tools that can combine R code, data, and text, producing reproducible and easily shared documentation.[8]

In summary, RStudio is a powerful and user-friendly development environment. It integrates various handy tools and features that can assist users in enhancing their development efficiency and code quality. Whether you're a beginner or a professional, you can effortlessly develop and manage R projects in RStudio. The download link for RStudio is: `https://posit.co/download/rstudio-desktop/`. This book recommends using R in conjunction with RStudio.

[7]For more details, see: `https://shiny.rstudio.com/`.
[8]For more information, see: `https://yihui.org/knitr/` and `https://rmarkdown.rstudio.com/`.

CHAPTER 2

R Programming Fundamentals

Many software developers find the design of the S language somewhat odd when they first encounter it. This is intrinsically related to the design philosophy of the S language. The creators of the S language aimed to develop a language suitable for both interactive data analysis and writing longer, more conventional programs. R, being a dialect of the S language, also inherited this characteristic. The RStudio development team once cited the words of the S language founder, John Chambers:

> "[W]e wanted to be able to begin in an interactive environment, where they did not consciously think of themselves as programming. Then as their needs become clearer and their sophistication increased, they should be able to slide gradually into programming, when the language and system aspects would become more important. [1]

In other words, the design philosophy of the S language is about the interaction between users and software developers, perhaps that is why some also refer to R as a weakly-typed language.

> **"Strongly-typed" vs. "weakly-typed" programming languages.** In programming, the terms "strongly-typed" and "weakly-typed" (or loosely-typed) refer to the extent to which a language enforces type checking rules. This is essentially about how strict a language is about mixing or converting types without explicit instructions from the programmer. A "weakly-typed" language, like R, is more permissive in allowing operations on data that might not be strictly appropriate or safe in terms of their data type. In these languages, implicit type conversion (often called type coercion) can happen. For instance, you might be able to add a string and a number without an error, with the language trying its best to make sense of the operation.

In this chapter, we will introduce foundational concepts of R programming. It begins with preparatory steps and progresses through basic mathematical operations, the intricacies of loops, and the creation and use of functions. The chapter then transitions into managing data with a focus on data structures, importing external data objects, and techniques for visualizing and analyzing data, concluding with guidance on utilizing R's "Help" feature.

1. Getting ready!

Now with a bit of background in the history of R, we are ready to start our journey! We need to first install R and RStudio. Start by visiting the CRAN website at `https://cran.r-project.org/`. Depending on your operating system (OS), select the appropriate version: Windows, macOS, or Linux (Debian, Fedora/Redhat, Ubuntu). For Windows users, click on the "base" link followed by `Download R for Windows`. macOS users should select `Download R for macOS`, while Linux users need to choose their specific distribution. After downloading, run the installer and follow the on-screen instructions.

Subsequently, for RStudio, visit its official website at `https://rstudio.com/products/rstudio/download/`. Again, based on your OS, select the corresponding version, such as `RSTUDIO-2023.06.1-524.EXE` for Windows 10/11, macOS 11+, or Ubuntu 20/Debian 11. After the download completes, execute the installer and complete the installation process. Once RStudio is installed, you can open it, and it will automatically detect your R installation. It's essential to note that R must be installed before RStudio since the latter relies on the former for its functionalities. Remember, R must be installed before you install RStudio because RStudio relies on R for its operations.

2. Basic Mathematical Operations and Fundamental Constants in R

2.1. Simple Math Operations. To experiment with basic arithmetic operations in R, enter the following commands in the R Console. Skip the comments, and after each command, press "Enter" to observe the results.

```
# Basic Arithmetic in R
# Addition
> 8 + 4
[1] 12
# Subtraction
> 9 - 2
[1] 7
# Multiplication
> 7 * 6
[1] 42
# Division
> 120 / 40
[1] 3
# Square root
> sqrt(64)
[1] 8
# Exponents
```

```
10^2
[1] 100
```

Below is an example illustrating R being a "weakly typed" language.

```
# Addition of a number with an array
> 3 + as.array(1)
[1] 4
# Inspecting the structure of an array
> str(as.array(1))
num [1(1d)] 1
# Inspecting the structure of a number
> str(1)
num 1
```

In R, the `str` function provides a compact display of the internal structure of an R object. It is particularly useful for getting a quick overview of more complex objects like data frames, lists, or models. For instance, if you have a data frame with several columns of different data types (e.g., numeric, factor, character), the `str` function will provide a summary showing the names of each column, the data type, and the first few entries in each column.

2.2. Mathematical Constants. In R, several mathematical constants are predefined for the convenience of the user. One of the most renowned constants, Euler's number, denoted as e, serves as the base for natural logarithms. In R, this constant can be derived using the `exp` function, such that `exp(1)` computes e^1, effectively yielding e. Another prominent constant is π, which signifies the ratio of a circle's circumference to its diameter. Conveniently, `R` provides a built-in command for this: simply typing `pi` returns the value of π. Moreover, R incorporates representations for positive and negative infinity through `Inf` and `-Inf`, respectively. Lastly, the result of undefined mathematical operations, like $\frac{0}{0}$, is termed as `NaN`, standing for "Not a Number" in R.

3. Loops in R

R excels at handling repetitive tasks. To execute a series of operations multiple times, we utilize loops. With a loop, R will carry out the instructions either for a designated number of iterations or until a certain condition is fulfilled. There are three primary loop types in R: the `for` loop, the `while` loop, and the `repeat` loop.

Loops are foundational in nearly all programming languages, including R. They offer a robust tool for automating tasks. However, it's worth noting that some believe they are sometimes overused in R programming.

3.1. For Loop. The `for` loop is a commonly used tool in R for repetitive tasks. Below is a simple example:

```
for (i in 1:5) {
print(i)
}
```

This loop uses an index variable `i` which gets assigned values in the sequence `1:10`. It's conventional to use `i` in loops, standing for "iteration", but any variable name can be used.

The counter inside the loop can also be modified:

```
for (i in 1:5 ) {
print(i + 1)
}
```

Below is a break-down of its functionality:

(1) Initializes a `for` loop where the looping variable, `i`, takes on values from the sequence 1 to 10.
(2) Inside the loop, for each value of `i`, the expression `i + 1` is evaluated.
(3) The result of the expression, which is the current value of `i` incremented by 1, is printed to the console.
(4) The loop continues this process until `i` has taken on all values in the sequence from 1 to 10.

3.2. While Loop. Another type of loop in R is the `while` loop. Unlike the `for` loop, which iterates over a predefined sequence, the `while` loop continues its execution as long as a specific logical condition remains true.

The basic structure of the `while` loop is as follows.

```
while(logical_condition) expression
```

Below is a simple example.

```
i <- 0
while (i <= 5) {
i <- i + 1
print(i)
}
```

Here's the breakdown of its functionality:

(1) Initialize a variable `i` with a value of 0.

(2) The `while` loop checks if the value of `i` is less than or equal to 5.
(3) If the condition `(i <= 5)` is true, the code inside the curly braces `{}` will execute.
(4) Inside the loop, `i` is incremented by 1 using the expression `i <- i + 1`.
(5) The updated value of `i` is then printed using the `print(i)` command.
(6) After printing, it returns to the beginning of the `while` loop to check the condition again. If `i` is still less than or equal to 5, it repeats the process; otherwise, it exits the loop.

Loops in R, while versatile, are often less efficient than using functions, especially with large datasets. However, they are irreplaceable for specific tasks such as simulations, recursive relationships, handling intricate problems with multiple conditions, or situations necessitating repeated checks like the "while" loops.

3.3. Repeat Loop. Another loop mechanism in R is the `repeat` loop. Unlike the `for` and `while` loops which have a conditional test at the start, the `repeat` loop will keep executing its body indefinitely. To stop its execution, a break statement is usually used inside the loop based on some logical condition.

The basic structure of the repeat loop is as follows.

```
repeat { expression }
```

Below is a simple example using the `repeat` loop.

```
i <- 0
repeat {
i <- i + 1
if (i > 5) {
break
}
print(i)
}
```

Here's the breakdown of its functionality:

(1) Initialize a variable `i` with a value of 0.
(2) Enter the repeat loop which doesn't have any initial conditional check.
(3) Inside the loop, increment `i` by 1 using the expression `i <- i + 1`.
(4) Check if the value of `i` is greater than 5 using the if condition.
(5) If the condition `(i > 5)` is true, the break statement will execute, terminating the loop.
(6) If the break statement is not executed, the value of `i` will be printed using the `print(i)` command.
(7) It will then loop back to the beginning of the repeat loop without any condition check and continue its execution.

3.4. Alternatives to Loops. In R, it is often recommended to use the *apply* family of functions; including `apply()`, `lapply()`, `tapply()`, `sapply()`, `vapply()`, and `mapply()`. These functions can typically execute tasks commonly done by loops, sometimes faster, and most importantly with a reduced risk of errors. Whenever you create a loop, consider remaking it using one of the apply functions. If feasible, opt for the `apply` version. It is crucial to reduce the risk of small errors in loops, which can potentially magnify into larger issues later on.

lapply. Your primary apply function will likely be `lapply()`, especially when you're starting out. The function `lapply()` iterates over each element in a list and executes a specified task or function. Additionally, it conveniently returns the results as a list, which would typically require extra coding when using a loop.

The structure of `lapply()` is:

```
lapply(X, FUN)
```

Here, `X` represents the vector you want to manipulate, and `FUN` stands for the function you want to apply.

For a straightforward demonstration, let's use `lapply()` to create a sequence from 1 to 5 and add 1 to each element, mirroring our previous for-loop approach:

```
lapply(0:4, function(a) {a + 1})
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
...
## [[5]]
## [1] 5
```

Do note that we specified our sequence as 0:4 to get the desired output. You can experiment with different sequences to see varying results.

Alternatively, you can define the function separately and then apply it using `lapply()`:

```
add_fun <- function(a) {a + 1}
lapply(0:4, add_fun)
```

The output remains the same.

sapply. The `sapply()` function behaves similarly to `lapply()`, but instead of returning a list, it outputs a vector:

```
sapply(0:4, function(a) {a + 1})
## [1] 1 2 3 4 5
```

Both methods produce results equivalent to those obtained using a for loop.

3.5. Functions in R. In R, a function is a named sequence of statements that takes a set of inputs, performs specific computations on these inputs, and returns an output. Functions are fundamental building blocks of R: they encapsulate logic, facilitate code reuse, and allow for abstraction.

In R programming, functions play a crucial role. There are numerous built-in functions available, such as `mean()`, `sum()`, `print()`, and `lm()`. Beyond these predefined functions, users have the flexibility to define their own functions tailored to their unique needs. Especially for statisticians, who often encounter unique analytical challenges or novel datasets, the ability to design bespoke algorithms is indispensable.

The typical syntax for defining a function in R is `function_name <- function(arg1, arg2, …) { ... return(result) }`, where the `return` statement is optional. By default, the function returns the last evaluated expression in its body. Function arguments in R are versatile. They can be numerous, have default values, and can be matched either by position or by name when calling the function.

One of the salient features of functions in R is their local environment. Variables created inside a function won't interfere with the global environment or any other function's environment. Furthermore, R functions have the property of closure, which means they can capture and utilize variables from the environment in which they were defined. This blending of local and defining environment variables makes R functions exceptionally potent. Lastly, R supports the paradigms of functional programming. This means functions can be passed as arguments, returned as values, or even stored within data structures, enhancing the flexibility and power of the language.

Below is a simple example for a function.

```
# Function Definition in R
add_numbers <- function(a, b) {
result <- a + b
return(result) }

# Using the Function
> sum_result <- add_numbers(5, 3)
[1] 8
```

The example provided above is undoubtedly what we would term a "toy example", as there's typically no need to write a function solely for adding two numbers.

Let's explore another more interesting example, writing a function to approximate the Euler's number e,.

The discovery of this remarkable constant is credited to the Swiss mathematician Jacob Bernoulli during his explorations of compound interest in the late 17th century. To illustrate Bernoulli's discovery in a practical context, let's consider an account initially holding £1.00 that accrues an annual interest rate of 20%. If the interest is applied once at the year's end, the total value of the account would increase to £1.20. However, the account's yield can be significantly impacted when we consider the effects of more frequent compounding of interest.

In a scenario where the interest is compounded biannually, the semiannual interest rate would be 10%, and hence the initial £1 gets multiplied by 1.1 twice, giving us £1.00 $\times$ 1.1^2 = £1.21 by the end of the year. With quarterly compounding, we have £1.00 $\times$ 1.05^4 = £1.21550625, and for monthly compounding, we have £1.00 $\times \left(1 + \frac{1}{12}\right)^{12}$ = £1.219391.... If we denote the number of compounding intervals by n, then each interval's interest would be $20\%/n$, and the end-of-year value becomes £1.00 $\times \left(1 + \frac{1}{n}\right)^n$.

Bernoulli noticed that as n becomes larger and the compounding intervals become smaller, this sequence approaches a certain limit. For example, compounding weekly ($n = 52$) gives £1.220198..., and compounding daily ($n = 365$) gives £1.220551... (roughly five pence more). As n grows towards infinity, this limit becomes the number known as e. Therefore, with continuous compounding, the account value will reach approximately £1.221402759...

However, it wasn't until the early 18th century that the number "e" was explicitly mentioned and popularized by Leonhard Euler. Because Euler was the first to identify and elucidate the unique properties of e that establish it as an **irrational number**.[2]

The constant e intriguingly emerges as the limit of the expression $(1 + 1/n)^n$ as n approaches infinity, a mathematical form that corresponds to the calculation of compound interest. Furthermore, e can also be characterized as the sum of an infinite series:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = 1 + \frac{1}{1} + \frac{1}{1 \cdot 2} + \frac{1}{1 \cdot 2 \cdot 3} + \cdots$$

Thus, we will create three functions, the first one will approximate e as the limit of the expression $(1 + 1/n)^n$ as n approaches infinity, the second and the third functions will approximate e as the sum of the infinite series $\sum_{n=0}^{\infty} \frac{1}{n!}$. The R codes are as follows.

```
# Approximating e, Euler's constant in R # First, using compound interest rate
e_fcn_1 <- function(n=2000){
(1+1/n)^n
}
e_fcn_1(1)
e_fcn_1(100)
e_fcn_1(10000)     # As n increases, we get a closer approximation to e
exp(1)     # This returns the value of Euler's constant, e

# Second, approximating e using the sum of an infinite series
e_fcn_2 <- function(n=2000){
e <- 0
for (i in 0: n){
e <- e + 1/factorial(i)
}
return(e)
}
e_fcn_2(1)
e_fcn_2(10)

# Third, a more concise version of the second method
e_fcn_3 <- function(n=2000){
e<- sum(1/factorial(0:n))
return(e)
}
e_fcn_3(1)
e_fcn_3(10)
```

4. Data Structures

R has several fundamental data types, including character, numeric, integer, complex, and logical. Alongside these, R also offers essential data structures such as vectors, lists, matrices, data frames, and factors. Some of these structures, like vectors and matrices, mandate that all their elements share the same data type. In contrast, lists and data frames can accommodate multiple data types. Additionally, objects in R can possess attributes, with examples being name, dimension, and class.

4.1. Vectors. A vector is an ordered collection of basic data types of a given length. The only key thing here is all the elements of a vector must be of the identical data type e.g homogeneous data structures. Vectors are one-dimensional data structures.

```
# R program to illustrate Vector
# Vectors (ordered collection of the same data type)
X = c(2, 4, 6, 8, 10)
# Printing those elements in the console
print(X)
```

Note that in R code, the # symbol is used to indicate a comment. Comments are lines of text that are not executed as part of the program but are included to provide explanations or annotations for the code. They are used to make the code more understandable to both the programmer and others who might read the code.

4.2. Lists. A list is a flexible container that holds a bunch of different things in a specific order. Lists can have all sorts of items mixed together. They're like organized rows where you can put vectors, matrices, words, functions, and more.

```
# R program to illustrate a List
# Employee IDs
ids = c(101, 102, 103, 104)
# Employee names
names = c("Alice", "Bob", "Carol", "David")
# Number of employees
totalEmployees = 4
# Combining data into a list
employeeList = list(EmployeeIDs = ids, Names = names, TotalEmployees =
totalEmployees)
# Displaying the employee list
print(employeeList)
```

4.3. Dataframes. A data frame is a two-dimensional data structure designed to store data in a tabular format. Similar to a table, data frames consist of rows and columns, and each column can hold a different vector of data. This allows for versatility in storing various types of information within a single data structure. In R programming, data frames are widely utilized, because they can hold diverse types of data, making them adaptable for a wide range of applications.

Key characteristics of dataframes include:

- Each dataframe must have rows and columns, forming a tabular layout.

- Different columns within a data frame can hold different vectors, enabling the storage of various data types.
- Each column must have the same number of elements, ensuring consistent alignment of data across the data frame.
- Within a single column, all elements must share the same data type for consistency.
- Notably, distinct columns within the same dataframe can hold different data types, providing flexibility for data representation and manipulation.

```
# Create a dataframe
myDataFrame <- data.frame (
Name = c("Alice", "Bob", "Carol"),
Age = c(28, 15, 35),
IsAdult = c(TRUE, FALSE, TRUE)
)
print(myDataFrame)
```

4.4. Matrices. A matrix is a rectangular arrangement of numbers organized in rows and columns. Rows run horizontally while columns run vertically. Matrices are homogeneous and two-dimensional structures.

To define a matrix in R, use the `matrix` function. It requires the elements you wish to include, the desired number of rows and columns. By default, matrices are populated column-wise.

```
# Defining a matrix in R
M <- matrix(
c(1, 2, 3, 4, 5, 6, 7, 8, 9),
nrow = 3, ncol = 3,
byrow = TRUE
)
print(M)
```

4.5. Arrays. Arrays in R can store data in more than two dimensions. For instance, an array with the dimensions (2, 3, 3) would produce three matrices, each having 2 rows and 3 columns. Arrays maintain homogeneous data.

To define an array in R, use the `array` function. It takes in the elements and the desired dimensions.

```
# Defining an array in R
B <- array(
c(1, 2, 3, 4, 5, 6, 7, 8),
```

```
dim = c(2, 2, 2)
)
print(B)
```

4.6. Factors. Factors categorize data into levels and are effective for storing categorical data, like strings and integers. They are commonly used in statistical modeling and data analysis.

To define a factor in R, use the `factor` function, which takes in a vector.

```
# Defining a factor in R
fctr <- factor(
c("Male", "Female", "Male", "Male", "Female", "Male", "Female")
)
print(fctr)
```

Haven't you noticed that we used "`<-`" and "`=`" interchangeably here? In R, the symbol "`<-`" is an assignment operator. It's used to assign values to variables. The arrow points to the object being assigned a value. R also supports the use of "`=`" for assignments. However, "`<-`" is more traditional and is often preferred by many R users for clarity, especially when distinguishing between assignment and function arguments.

5. Importing Data Objects

We don't always input data using the R console, especially when dealing with large datasets. For such cases, it's important to know how to load specific datasets in R, such as `mdeaths`, into R. The `mdeaths` dataset presents a time series of monthly male deaths from bronchitis, emphysema, and asthma in the UK, specifically in males, from 1974 to 1979 [5]. The data is stored as `csv` format.

What is a time series? A time series is a sequence of numerical data points taken at successive, usually equally spaced, points in time. It is used to track changes over time and can represent anything that varies chronologically such as stock prices, temperature records, and monthly sales data.

Here `csv` stands for "Comma-Separated Values". It is a simple file format used to store tabular data, such as a spreadsheet or a database. Each line of the file represents a row of data, and within each line, columns or fields are separated by commas. R boasts remarkable versatility in importing a wide array of data formats. For instance, users can directly import plain text files such as comma-separated values (CSV) using functions like `read.csv()`. Excel files, both in XLS and XLSX formats, can be effortlessly read using the `readxl` package. For those transitioning from

other statistical platforms, R also offers compatibility with proprietary formats, such as SPSS files, through the `haven` package.

What is an R package? An R package is a collection of bundled functions, data, and documentation designed to perform specific tasks in the R programming environment. These packages extend R's base functionality, allowing users to accomplish a wide range of specialized tasks. They may contain tools for data visualization, like `ggplot2`, or data manipulation, like `dplyr`. Packages are typically hosted on the CRAN, making it easy for users to install and implement them. Once installed, a package is loaded into the R session using the `library()` function, granting access to its features.

Below is how we import the `mdeaths` dataset into R.

```
# Setting the working directory in R
folder <- "/Users/sunjiajing/R-DPS/CH2"
setwd(folder)
# Loading the mdeaths dataset
mdeaths <- read.csv("mdeaths.csv")
head(mdeaths)
```

You need to replace `/Users/sunjiajing/R-DPS/CH2` with your own working directory. Note that in R, the function `head` is used to display the first few rows (by default, the first six rows) of a data object, such as a data frame or a vector. This function is especially useful when dealing with large datasets, as it allows users to quickly get a glimpse of the structure and some sample values without viewing the entire dataset. You can also check the structure of `mdeaths` dataset, by inputting `str(mdeaths)`.

The output is as follows.

```
  X   x
1 1 2134
2 2 1863
3 3 1877
4 4 1877
5 5 1492
6 6 1249
```

We didn't have to import the `mdeaths` dataset. In fact, the `mdeaths` is conveniently bundled within R's core package `datasets`. Because `datasets` package is one of the base R packages, you don't have to install it separately.

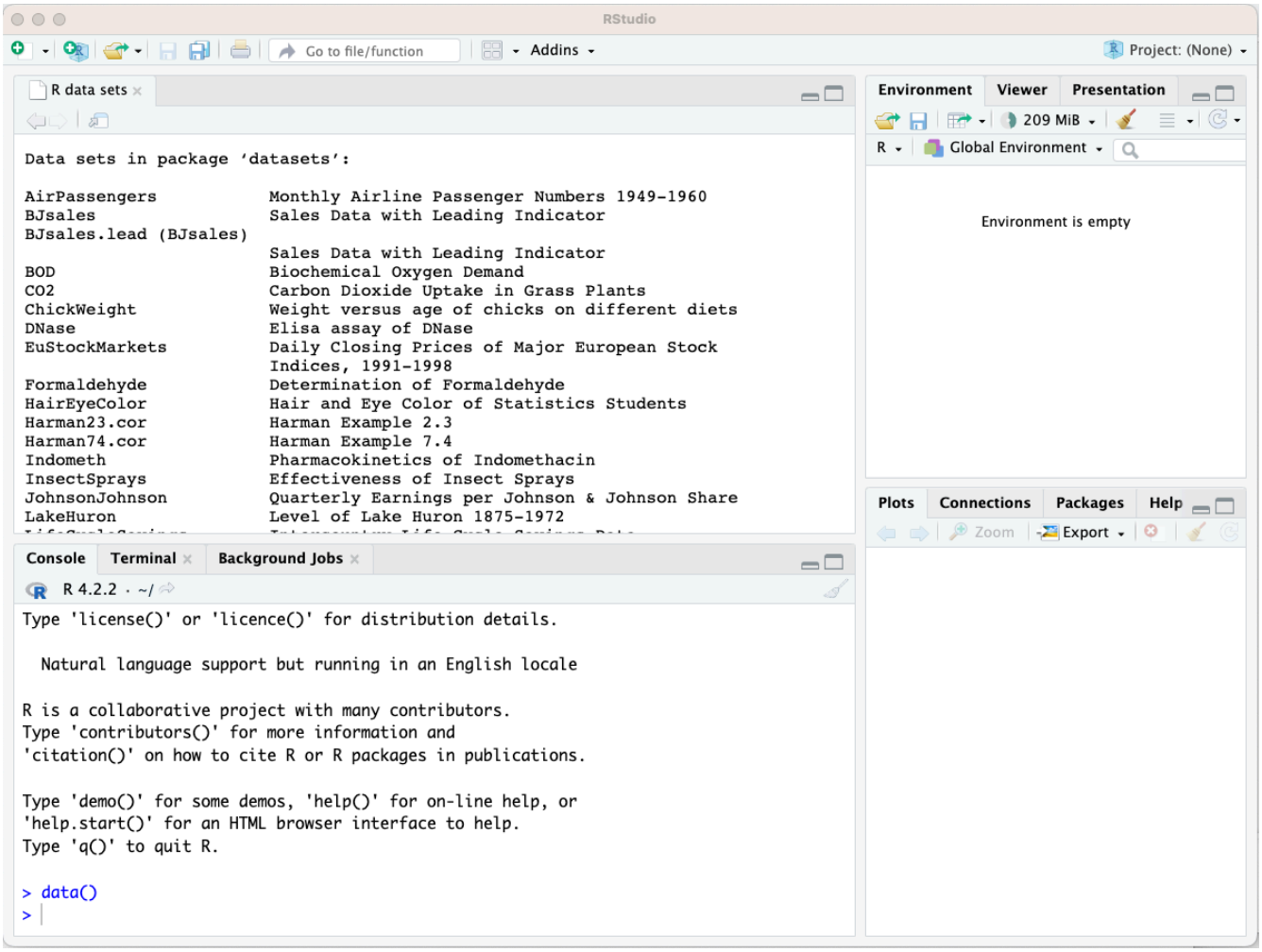


FIGURE 1. List of pre-loaded data

What is an R base package? An R base package refers to the set of foundational packages that come pre-installed with every R installation. These packages provide the essential functions and capabilities that allow R to operate as a powerful statistical programming language.

Below, we will describe how to load and use these built-in datasets. To see the list of pre-loaded data, type the function `data()`:

```
data()
```

So all we need to do is as follows.

```
library(datasets)
head(mdeaths)
```

Here `head(mdeaths)` displays the first few records of the `mdeaths` dataset.

6. Basic Dataset Manipulations in R

R provides a comprehensive suite of functions and packages for data manipulation. Among the most fundamental operations are *subsetting*, *filtering*, and *aggregating*. These operations often serve as the initial steps in any data analysis pipeline, ensuring that the data is structured appropriately for subsequent analyses.

6.1. Subsetting. Subsetting in R involves extracting parts of your data set. This can be done for both rows and columns if the dataset is matrix, or for elements if the dataset is a vector

```
 # Example: Selecting the 2nd to 4th elements
subset_mdeaths <-mdeaths$x [2:4]
```

6.2. Filtering. Filtering is a process by which rows are selected based on certain conditions. This operation is particularly helpful in refining the dataset to a more relevant or focused subset.

```
 # Example: Filtering elements where the the number of death is above 1500
filtered_mdeaths <- mdeaths$x [mdeaths$x > 1500 ]
```

6.3. Aggregating. Aggregation in R involves summarizing large chunks of data into simpler metrics. Common aggregations include computing the mean, sum, or count of a particular column based on groupings in another column. So let's use the `mtcars` dataset available in base R as an example. This dataset comprises various car models and their specifications. It originally comes from the 1974 Motor Trend US magazine and comprises fuel consumption and 10 other aspects of automobile design and performance for 32 automobiles (1973–74 models).

- **Using the aggregate function:** This function is part of base R and allows data to be aggregated using a formula.
- **Using dplyr:** The `dplyr` package offers a more sophisticated and readable syntax for data aggregation using functions like `group_by` and `summarise`.

```
# Load the necessary library
library(dplyr)

# Compute the aggregation
avg_mpg_by_cyl <- mtcars %>%
group_by(cyl) %>%
```

```
summarise(avg_mpg = mean(mpg))

# Display the result
print(avg_mpg_by_cyl)
```

The result will display the average `mpg` for cars with 4, 6, and 8 cylinders, offering insights into fuel efficiency based on engine size. This kind of aggregation provides a summarized view of larger datasets, facilitating easier analysis and decision-making.

In conclusion, mastering these fundamental data manipulation operations is crucial for any data analyst or scientist working with R. They serve as the foundational blocks upon which more complex operations and analyses can be built.

7. Data Visualization in R

Visualizing data in R, or in any other platform, is fundamentally crucial for data analysis. A simple visualization can immediately uncover patterns and outliers not evident in raw data, making comprehension more intuitive. Beyond personal understanding, these visual representations enable clearer communication of complex data points to diverse audiences. This clarity further streamlines the decision-making process, allowing stakeholders to discern trends and correlations essential for strategic actions. In short, R's visualization capabilities transform raw data into actionable insights.

7.1. Basic Line Plot. We can use the following command to plot the monthly deaths dataset.

```
# Visualization of the data in R using a blue line graph
plot(mdeaths$x, type="l", col="blue")
```

Note that in R, within the context of a dataframe, the $ operator is used to access specific columns by name. It allows you to reference a column of a dataframe as a vector. The plot is visualized in Figure 2.

7.2. Improve Line Plots. A closer look at Figure 2 reveals a lack of clarity. For example, the series represents monthly deaths from bronchitis, emphysema, and asthma for males in the UK between 1974–1979. However, the horizontal axis does not display the time indices, and the vertical axis lacks a clear label. Perhaps a main title for the figure would be more appropriate instead of the vertical label.

This can be achieved by converging the male deaths into a time series object, using the `ts()`, which is a function in the base package `stats`. Further customization can then be achieved by adding more parameters to the `plot` function. The R code is as follows.

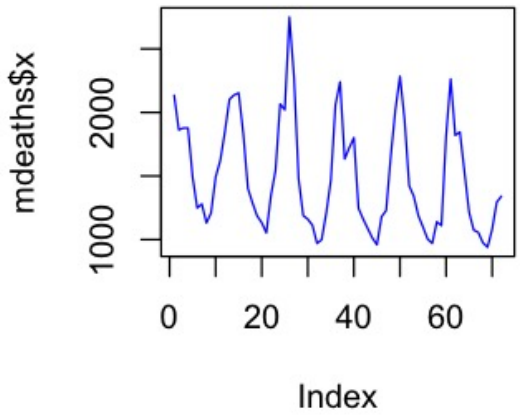


FIGURE 2. Monthly deaths from bronchitis, emphysema, and asthma for males in the UK between 1974–1979.

```
# Convert mdeaths$x into a time series starting from January 1974
mdeaths_ts <- ts(mdeaths$x, start=c(1974, 1), frequency=12)

# Plot the mdeaths time series with specified attributes
plot(mdeaths_ts, type="l", col="blue", xlab ="Time", ylab="",
main = "Monthly deaths between 1974-1979")
```

Here is the explanation of the R commands.

- `mdeaths_ts <-`: This assigns the result of the `ts()` function to a new variable named `mdeaths_ts`.
- `ts()`: Creates a time series object in R.
  - `mdeaths$x`: Specifies the data to convert into a time series, extracting the column named `x` from the `mdeaths` dataset.
  - `start=c(1974, 1)`: Defines the start time of the series as January of 1974.
  - `frequency=12`: Indicates there are 12 observations per year, suggesting monthly data.

The command `plot(mdeaths_ts, type="l", col="blue", xlab="Time", ylab="", main="Monthly deaths between 1974-1979")` visualizes the time series:

- `plot()`: Visualizes data in R, producing a time series plot given a time series object.
- `type="l"`: Specifies a line plot.
- `col="blue"`: Sets the line color to blue.
- `xlab="Time"`: Labels the x-axis as ”Time”.
- `ylab=""`: No label is set for the y-axis.

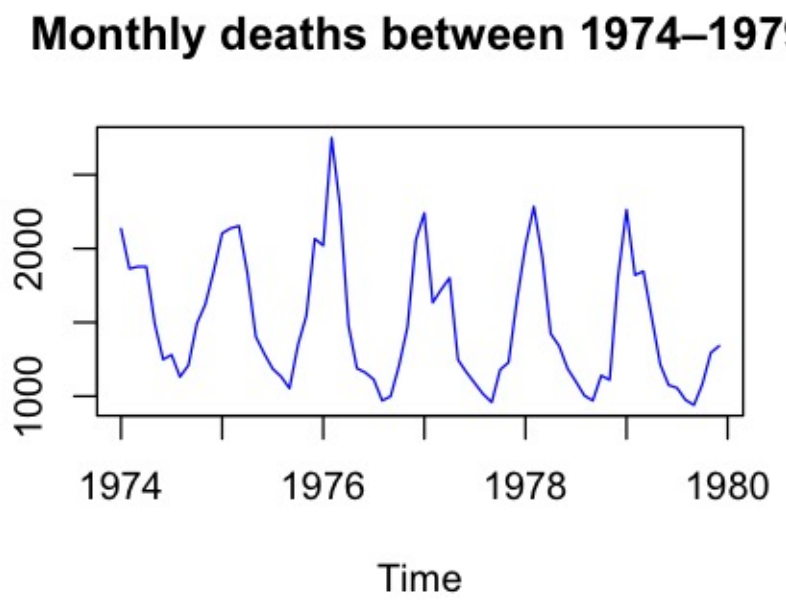


FIGURE 3. Monthly deaths from bronchitis, emphysema, and asthma for males in the UK between 1974–1979.

- `main="Monthly deaths between 1974-1979"`: Sets the main title of the plot.

The improved visualization can be seen in Figure 3, which is a significant improvement over Figure 2.

7.3. Other R Plots. Other than line plot, R offers a wide range of plotting capabilities, both through its base graphics system and various packages. Below are some common types of plots available in R's base graphics and the associated functions.

- **Scatterplot**: Plots points based on two variables. Use the `plot()` function.
- **Histogram**: Represents the distribution of a single variable by binning values and counting the number of observations in each bin. Use the `hist()` function.
- **Boxplot**: Displays the distribution's five-number summary (minimum, first quartile, median, third quartile, and maximum). Useful for comparing distributions across groups. Use the `boxplot()` function.
- **Bar Plot**: Displays values (often frequencies) as bars. Use the `barplot()` function.
- **Pie Chart**: Represents proportions or percentages as slices of a pie. Use the `pie()` function.
- **Density Plot**: Estimates and plots the probability density function of a variable. Use the `plot(density(data))` approach. In Section 5, we have superimpose a density plot over the histogram.
- **Pairs Plot (Scatterplot Matrix)**: Plots pairwise relationships between multiple variables. Use the `pairs()` function.

- **QQ-Plot (Quantile-Quantile Plot)**: Compares two probability distributions by plotting their quantiles against each other. Use the `qqnorm()` and `qqline()` functions.
- **Contour Plot**: Represents three-dimensional data in two dimensions using contour lines. Typically used for plotting a response variable as a function of two predictors. Use the `contour()` function.
- **Image Plot**: Like a contour plot but colors regions between contour lines. Use the `image()` function.
- **3D Scatter Plot and Surface Plots**: These are not directly available in base graphics, but can be created using specialized packages like `rgl` and `lattice`.

Exercise: You are encouraged, in your free time, to try these R commands.

The `mtcars` dataset is versatile and contains various attributes of car models, which makes it suitable for a variety of plots and visualizations in R. Here are some potential plots/visualizations using the `mtcars` dataset.

Histogram:

```
hist(mtcars$mpg, main="Histogram of Miles-per-Gallon",
xlab="mpg", col="lightblue", border="black")
```

Scatter plots:

```
plot(mtcars$wt, mtcars$mpg, main="Weight vs. MPG",
xlab="Weight", ylab="MPG", pch=19, col="blue")
```

Box plots:

```
boxplot(mpg ~ am, data=mtcars, main="MPG by Transmission Type",
xlab="Transmission (0=Automatic, 1=Manual)", ylab="MPG",
col=c("lightblue","lightgreen"))
```

Par plots (scatterplot matrix):

```
pairs(~mpg+hp+wt+qsec, data=mtcars, main="Scatterplot Matrix")
```

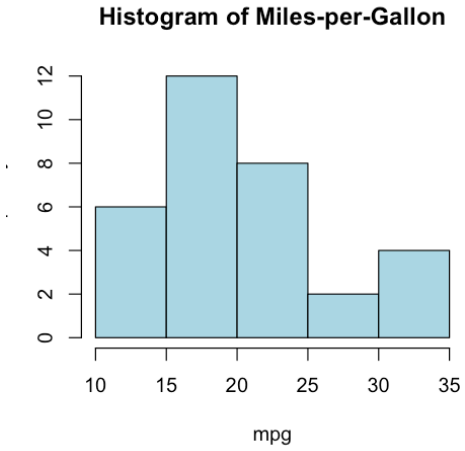
Correlation heatmap:

```
library(corrplot)
correlations <- cor(mtcars)
corrplot(correlations, method="circle")
```

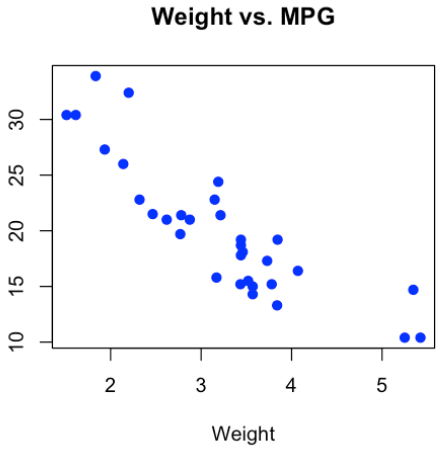
3D scatter plots:

```
library(scatterplot3d)
scatterplot3d(mtcars$hp, mtcars$wt, mtcars$mpg,
pch=19, color="blue", main="3D Scatterplot: HP vs. Weight vs. MPG")
```

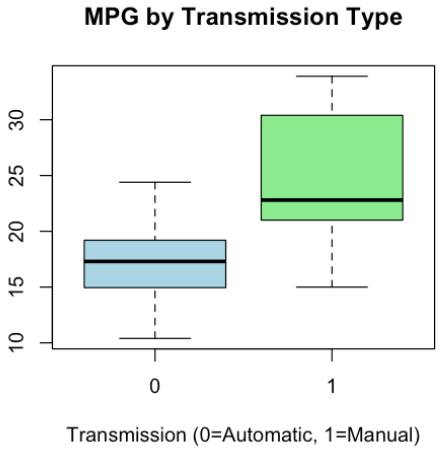
The plots are visualized in Figure 4.



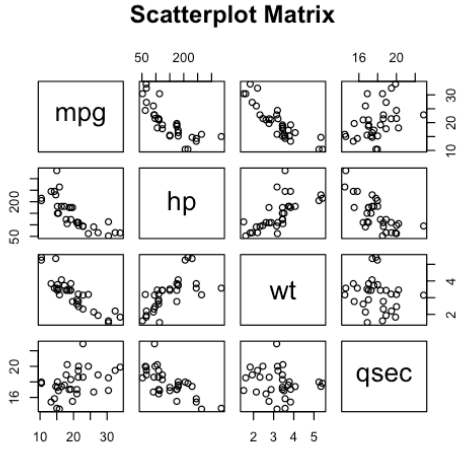
(A) Histogram



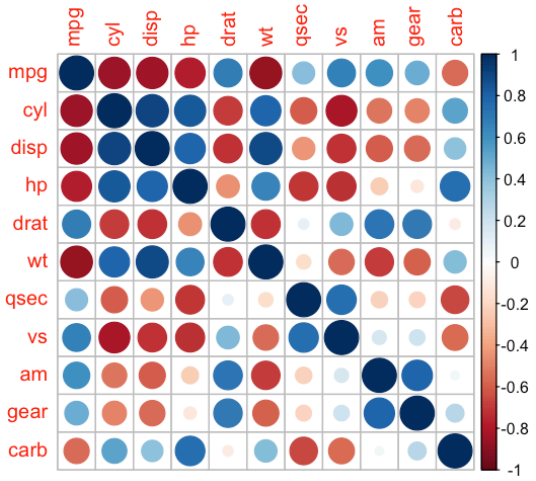
(B) Scatter plot



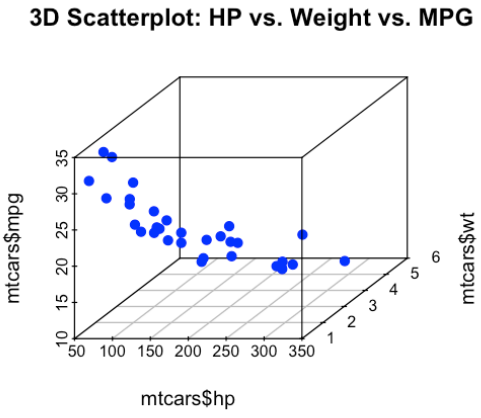
(C) Box plot



(D) Pair plot



(E) Correlation



(F) 3D plot

FIGURE 4. Various visualizations of the `mtcars` dataset

8. Preliminary Data Analysis

For now, let's inspect some statistical properties of the dataset, specifically, the mean and variance of monthly deaths from bronchitis, emphysema, and asthma in the UK.

For those of you who need a refresher on the concepts of mean (expected value) and variance, recall that for a discrete random variable X with possible values $x_1, x_2, \ldots$ and associated probabilities $p(x_1), p(x_2), \ldots$:

- **Mean (Expected Value)**:

$$E(X) = \mu = \sum_i x_i \cdot p(x_i)$$

The mean, often represented by μ for a population, is a measure of the central tendency of a distribution. In practice, we often don't have access to the entire population data, so we use the **sample mean** to approximate the population mean. The sample mean is denoted as $\bar{x}$ and is calculated as:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

where n is the sample size.

- **Variance**:

$$\text{Var}(X) = \sigma^2 = \sum_i (x_i - \mu)^2 \cdot p(x_i)$$

Variance, represented by σ^2 for a population, measures the spread or dispersion of data points in a distribution. Similar to the mean, when we don't have population data, we use the **sample variance** to approximate the population variance. The sample variance, s^2, is computed as:

$$s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2$$

Note the denominator is $n-1$, which is a correction (Bessel's correction) to ensure an unbiased estimator for population variance.

- Moreover, because we typically do not know the population mean and variance in a practical context, we usually refer to them as the sample mean and variance when only mean and variance are mentioned.

The R code is spelled out as below.

```
# Calculating the mean of the x column from mdeaths dataset
avg_mdeaths <- mean(mdeaths$x)
```

```
# Calculating the variance of the x column from mdeaths dataset
var_mdeaths <- var(mdeaths$x)
```

The findings can be summarized as follows.

- The **mean** (average) monthly deaths is 1495.944. This suggests that, on average, approximately 1496 people died monthly due to these conditions during the period covered by the dataset.
- The **variance** of the monthly deaths is 187619.7. Variance quantifies how spread out the numbers in a dataset are. A higher variance indicates that the numbers are more spread out from the mean, while a lower variance indicates they are closer to the mean.
- The **standard deviation**, which is the square root of the variance, is approximately 433.1509. The standard deviation provides a measure of the amount of variation or dispersion in the set of values.

Note that random variables have various statistical properties that can be described using measures other than just the mean and variance.

- **Median**: For a random variable X, the median m is defined such that $P(X \leq m) \geq 0.5$ and $P(X \geq m) \geq 0.5$.
- **Mode**: The value that appears most frequently in a data set or the peak of the probability distribution.
- **Skewness**: A measure of the asymmetry of the probability distribution about its mean. If μ is the mean and σ is the standard deviation, then skewness is given by:

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

- **Kurtosis**: A measure of the "tailedness" of the probability distribution:

$$\text{Kurtosis}(X) = \mathbb{E}\left[\left(\frac{X-\mu}{\sigma}\right)^4\right] - 3$$

- **Moment**: The n^{th} moment about the origin of X is $\mathbb{E}[X^n]$. The n^{th} moment about the mean is $\mathbb{E}[(X-\mu)^n]$.
- **Quantiles and Percentiles**: For any $0 \leq p \leq 1$, the p-quantile of X is a value q such that $P(X \leq q) = p$.

In the R language, a plethora of functions and packages are available to evaluate the statistical properties of random variables. The function `median()` is employed to compute the median of a dataset. For determining the mode, there isn't a built-in R function, but one can easily create a custom function or utilize the `Mode` function from the `DescTools` package. The `skewness()` and `kurtosis()` functions from the `moments` package can be used to assess the skewness and kurtosis of a distribution, respectively. The `moment()` function, also from the `moments` package, is useful

for calculating raw and central moments. Lastly, to get quantiles or percentiles of a dataset, the built-in R function `quantile()` is very handy.

Exercise: Please calculate the statistical properties for the aforementioned dataset.

Quiz: Do all variables that follow a pre-specified distribution have a mean and variance?

The answer is no. Take the Cauchy distribution as an example, which is also referred to as the Lorentz distribution. The Cauchy distribution is named after the French mathematician Augustin-Louis Cauchy (1789–1857). Though whether he actually first discovered it was still of scientific debate. As discussed in Chapter 18 of [13], function with the form of the density function of the Cauchy distribution was studied geometrically by Fermat in 1659, and later became known as the "witch of Agnes", after Agnesi included it as an example in her 1748 calculus textbook. Despite its name, the first explicit analysis of the properties of the Cauchy distribution was published by the French mathematician Poisson in 1824, with Cauchy only becoming associated with it during an academic controversy in 1853.

Cauchy distribution is noteworthy because it does not have a defined mean or variance. Its probability density function (PDF) is given by:

$$f(x; x_0, \gamma) = \frac{1}{\pi\gamma \left[1 + \left(\frac{x-x_0}{\gamma}\right)^2\right]}$$

where x_0 represents the location parameter (essentially the median of the distribution) and γ is the scale parameter, which corresponds to half of the full width at half maximum or half the interquartile range. The Cauchy distribution looks similar to a bell curve in its shape, but its tails are longer than those of the normal distribution. Probabilities in the tails of the Cauchy distribution decrease rapidly, yet they remain nonzero, whereas the tails of the normal distribution taper off exponentially. The Cauchy distribution has found applications in various fields. In finance, for example, it can model stock price fluctuations. Meanwhile, in physics, it's utilized to describe particle energy loss.

9. Using the "Help" tab

Should you find yourself uncertain about the syntax of a command, such as `runif`, there's no need to fret. One efficient approach is to input the command into the "Help" tab, which acts as a portal to the vast R documentation. An illustration of this is depicted in Figure 5. For more direct access, consider using `help(runif)` or simply `?runif` in the console to pull up the function's comprehensive documentation.

By following the aforementioned steps, you'll be directed to the following resource: `https://stat.ethz.ch/R-manual/R-devel/library/stats/html/Uniform.html`. This webpage offers an insightful overview and details, a snippet of which is shared below:

Description

Density, distribution function, quantile function, and random generation for the uniform distribution on the interval from `min` to `max`.

Usage

```
dunif(x, min = 0, max = 1, log = FALSE)
punif(q, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE)
qunif(p, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE)
runif(n, min = 0, max = 1)
```

Arguments

`x, q`: vector of quantiles.
`p`: vector of probabilities.
`n`: number of observations. If `length(n) > 1`, the length is taken to be the number required.
`min, max`: lower and upper limits of the distribution. Must be finite.
`log, log.p`: logical; if TRUE, probabilities p are given as log(p).
`lower.tail`: logical; if TRUE (default), probabilities are $P[X \leq x]$ otherwise, $P[X > x]$.

Details

If `min` or `max` are not specified they assume the default values of 0 and 1, respectively.
The uniform distribution has density:

$$f(x) = \frac{1}{\texttt{max} - \texttt{min}}$$

for $\texttt{min} \leq x \leq \texttt{max}$.

Value

`dunif` gives the density, `punif` gives the distribution function, `qunif` gives the quantile function, and `runif` generates random deviates. ...

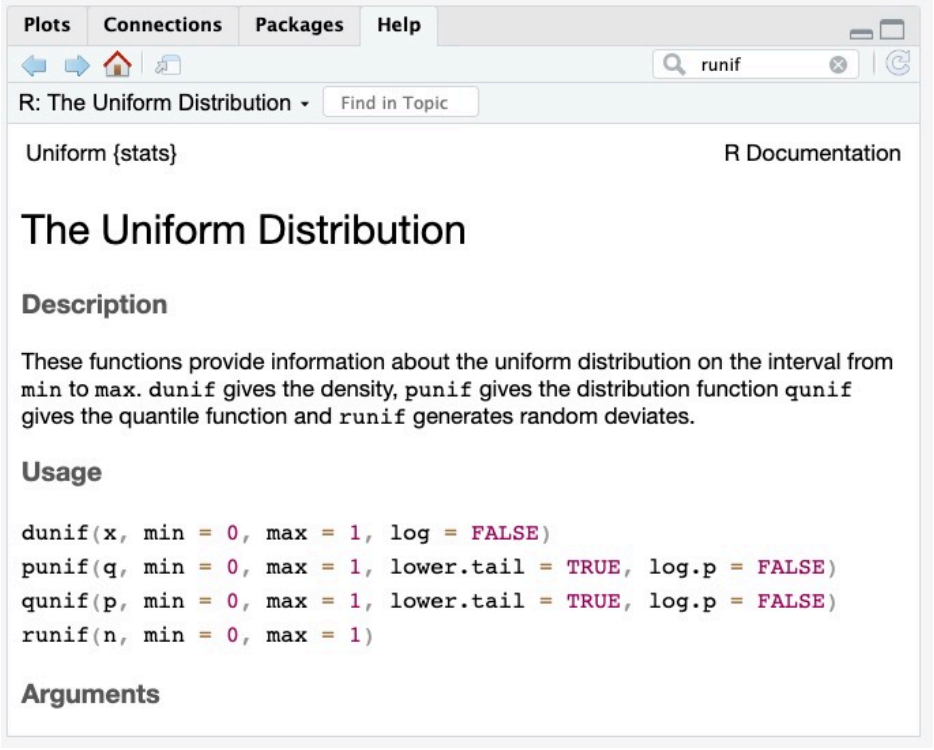


FIGURE 5. The "Help" tab

As you can see, the help page provides detailed information on the uniform distribution functions in the R programming language. It elaborates on functions such as `dunif`, `punif`, `qunif`, and `runif`, along with their respective parameters and options.

- `dunif`: Computes the density of the uniform distribution for a given set of values on the interval from `min` to `max`.
- `punif`: Computes the cumulative distribution function for the uniform distribution, representing the probability that a uniformly distributed random number will be less than a given value.
- `qunif`: Provides the quantile function for the uniform distribution, indicating the value below which a given percentage of observations in a group fall.
- `runif`: Generates random numbers from a uniform distribution between specified `min` and `max` values.

The help page also includes details about the underlying properties of the uniform distribution, considerations for the default parameters, and references to foundational literature and related functions in R.

CHAPTER 3

Probability Distributions

Here, we will begin by introducing how to code random experiments in R. We will then delve into generating random variables from various discrete and continuous distributions in R. Additionally, we will cover the inverse cumulative density function (CDF) method for complex distributions that do not have a built-in random generator in R.

1. Random Experiments and Random Variables

Certain natural scenarios come with an inherent element of randomness or unpredictability. Consider the act of rolling a dice, flipping a coin, or monitoring the Google share price. Our journey begins by delving into random experiments, which are fundamental processes that yield well-defined outcomes, albeit unpredictably. Building upon this, we'll explore how these experiments lead to the concept of a random variable. A random variable effectively quantifies the outcomes of these experiments, translating uncertain scenarios into numerical values. By understanding the relationship between random experiments and random variables, we can model, analyze, and predict uncertain situations in a structured and mathematical way.

Here we focus on coin tossing. The ritual of tossing a coin, an act as simple as it is steeped in history, has been a method for resolving indecision for millennia. Originating in ancient Rome, the practice was known as "heads and ships", paying homage to the two-headed god Janus and the ship designs on their coins. In Britain, this evolved to "cross and pile", inspired by the markings on local coinage. Today, the universally recognized "heads or tails" persists. Beyond being just a game of chance, the coin toss has become synonymous with impartiality and fairness, frequently determining the course of events in sports and other domains [14]. Because of its simplicity and intuitive nature, coin tossing is often used to introduce the concepts of randomness and probability.

1.1. Coin Object. Consider a standard coin with two distinct sides: head and tail, see Figure 1. To simulate a coin toss in R, we first need to construct an object that mimics a coin. How can we achieve this? The simplest approach to model a coin with its two sides, "heads" and "tails", is via a character vector using the `c()` function:

```
# A (virtual) coin object
coin <- c("heads", "tails")
coin
#> [1] "heads" "tails"
```

A numeric coin representing "1" for heads and "0" for tails can also be created:

FIGURE 1. Head and tail of a British penny

```
# A numeric coin object
num_coin <- c(1, 0)
num_coin
#> [1] 1 0
```

In fact, this numeric representation of a coin flip is an example of how a random variable quantifies the outcomes of random experiments. By assigning the value 0 to "tails" and 1 to "heads", the random variable translates the uncertain outcome of the coin flip into a numerical format. This is, in essence, the role of a random variable.

Additionally, a logical coin can be represented with "TRUE" for heads and "FALSE" for tails:

```
# A logical coin object
log_coin <- c(TRUE, FALSE)
log_coin
#> [1] TRUE FALSE
```

1.2. Tossing a Coin. After defining a coin in R, we can now simulate its toss.

```
# Simulating a coin toss
coin <- c('heads', 'tails')
sample(coin, size = 1)
#> [1] "heads"
```

By default, `sample()` selects without replacement. If the desired sample size exceeds the length of the input vector, sampling with replacement becomes necessary. It's also worth noting that we

allow sampling with replacement because each toss is independent of the others. The R code is as follows.

```
# Sampling with replacement
sample(coin, size = 4, replace = TRUE)
#> [1] "tails" "heads" "tails" "heads"
```

1.3. The Random Seed. To ensure reproducibility in random sampling, we need to set a random seed. In R, the rand seed gets determined using the `set.seed()` function, accompanied by an arbitrary integer:

```
# Setting the random seed
set.seed(1257)
# Tossing the coin with replacement
sample(coin, size = 4, replace = TRUE)
#> [1] "tails" "heads" "heads" "tails"
```

In R, a random seed, often referred to as "seed", is the initial input for generating a sequence of pseudorandom numbers. Note that while the numbers generated by computational systems appear random, they aren't genuinely random. They are determined by algorithms which, if provided the same initial input (or seed), will produce the same sequence of numbers each time. This is why they are called "pseudorandom" rather than truly random.

This deterministic nature of seeds is pivotal in computational research for three main reasons: First, to ensure **reproducibility**. Central to scientific research, reproducibility validates the results. In tasks like statistical analysis or machine learning where randomness is integral, a fixed seed guarantees that results can be replicated by anyone, anytime. Second, to maintain **consistency**. In procedures such as model tuning, a set seed ensures consistent outcomes across iterations, providing a reliable benchmark. Third, to facilitate **debugging**. When computational tasks yield unexpected behaviors, it can be challenging to trace the origins of such anomalies without a set seed. Setting a seed ensures any anomalies can be consistently reproduced, aiding in their resolution.

Exercises: Apart from coin tossing, how would you design the following random experiments in R?

(1) Selecting a card from a standard deck is an experiment.
(2) We choose three sweets together from a bag containing yellow, green, and red sweets.
(3) Throwing a dart at a dartboard.

2. Discrete Distributions

2.1. Discrete Uniform Distribution. It's important to understand that there are two main types of uniform distributions: the discrete uniform distribution and the continuous one. The distinctions between them are fairly evident from their respective names. The **Discrete Uniform Distribution** relates to a discrete random variable, indicating that the variable can assume only a finite or countably infinite set of values.[1]

DEFINITION 1. A random variable X is said to have a **discrete uniform distribution** over a finite set Ω with n elements if

$$P(X = x_i) = \frac{1}{n}$$

for every $x_i \in \Omega$.

In base R, there isn't a dedicated function to simulate a discrete uniform random variable, unlike other random variables such as Normal, Poisson, and Exponential. However, the `rdunif` function from the `purrr` package offers this capability. If you haven't previously installed the `purrr` package, use the command `install.packages("purrr")` to install it.

The syntax for the `rdunif` function is `rdunif(n, b, a)`, where `n` represents the number of random values to return, `b` denotes the maximum value of the distribution. It's required to be an integer due to the discrete nature of the distribution, and `a` indicates the minimum value of the distribution, and it should also be an integer.

To simulate 10 ages between 10 to 50, you can execute the following:

```
# Loading the purrr library
library(purrr)
# Simulating 10 ages between 10 to 50
rdunif(10, b=50, a=10)
#> [1] 37 25 31 46 18 14 47 25 13 43
```

For simulating 10 ages with a maximum age of 100, you can use:

```
# Simulating 10 ages up to 100
rdunif(10, 100)
#> [1] 86 90 70 79 78 14 56 62 4 4
```

If you wish to simulate the discrete uniform distribution for negative integers:

[1]Conversely, the **Continuous Uniform Distribution** pertains to a continuous random variable, suggesting that the variable can adopt any value within a designated range.

```
# Simulating the discrete uniform distribution for range -10 to 10
rdunif(10, 10, -10)
#> [1] 10 -3 9 -8 -7 -6 -9 4 -3 9
```

2.2. Bernoulli Distribution. At the core of many complex discrete distributions is the fundamental **Bernoulli Distribution**, named after the Swiss mathematician Jacob Bernoulli. This distribution pertains to experiments that have only two possible outcomes: success (often denoted as 1 or "yes") and failure (denoted as 0 or "no"). Every random experiment that leads to a yes/no, true/false, or success/failure outcome can be modeled as a Bernoulli trial. The Bernoulli distribution can be viewed as the building block for various other discrete distributions. For instance, flipping a coin, where one is interested in the event of landing heads, is a classic example of a Bernoulli trial.

DEFINITION 2. A random variable X is said to have a **Bernoulli distribution** with parameter p where $0 \leq p \leq 1$. Its probability mass function is given by:

$$P(X = k) = \begin{cases} p & \text{if } k = 1 \\ 1 - p & \text{if } k = 0 \end{cases}$$

where:

- p represents the probability of success (outcome 1).
- k can only take the values 0 or 1.

When $p = 0.5$, this distribution effectively models the outcome of tossing a fair coin. However, when $p \neq 0.5$, we often refer to it as an "unfair" coin toss.

In R, the `rbinom` function from the base package can be used to simulate values from the Bernoulli distribution. While this function is primarily meant for the binomial distribution, by setting the number of trials to one, it effectively models a Bernoulli trial. The syntax for this is `rbinom(n, size = 1, prob)`, where `n` indicates the number of observations and `prob` represents the probability of success.

For instance, to simulate 10 Bernoulli trials with a success probability of 0.4, you can employ:

```
# Simulating 10 Bernoulli trials with success probability 0.4
rbinom(10, size = 1, prob = 0.4)
#> [1] 1 0 1 0 1 0 0 1 0 1
```

Another scenario might involve simulating 20 Bernoulli trials where the outcome is true with a probability of 0.5 (equivalent to a fair coin toss):

```
# Simulating 20 fair Bernoulli trials
rbinom(20, size = 1, prob = 0.5)
#> [1] 1 0 0 1 0 1 1 1 0 0 1 0 1 0 1 1 0 1 1 0
```

2.3. Binomial Distribution. The **Binomial Distribution** characterizes the number of successes in a fixed number of Bernoulli trials with the same probability of success.[2]

DEFINITION 3. A random variable X is said to follow a **binomial distribution** with parameters n (number of trials) and p (probability of success on a single trial) if its probability mass function is given by:

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

for $k = 0, 1, 2, \ldots, n$.

In R, the binomial distribution can be readily simulated using the `rbinom` function, which is built into the base package. No additional installations are necessary. The syntax for the `rbinom` function is `rbinom(n, size, prob)`, where `n` represents the number of observations (or trials) to simulate, `size` denotes the number of trials in each observation, and `prob` is the probability of success in each trial.

To simulate 10 experiments, each with 5 trials and a success probability of 0.6, you can use:

```
# Simulating 10 experiments each of 5 trials with success probability 0.6
rbinom(10, size=5, prob=0.6)
#> [1] 3 4 2 5 3 4 3 2 4 2
```

For simulating 20 experiments, each with 10 trials and a success probability of 0.7, you can do:

```
# Simulating 20 experiments each of 10 trials with success probability 0.7
rbinom(20, size=10, prob=0.7)
#> [1] 7 6 8 7 7 8 9 8 7 6 7 8 7 6 8 7 7 8 9 8
```

2.4. Hypergeometric Distribution. Stepping further into discrete probability distributions, we encounter the **Hypergeometric Distribution**. It models the number of successes in a sample drawn without replacement from a population containing a fixed number of successes

[2]A Bernoulli trial is a random experiment in which there are only two possible outcomes: success or failure. The binomial distribution is used to model the probability of observing a specific number of successes in a given number of trials.

and failures. For instance, imagine you're drawing cards from a well-shuffled deck and you want to know the probability of drawing a specific number of face cards without replacing them.[3]

DEFINITION 4. A random variable X is said to have a **hypergeometric distribution** if its probability mass function is given by:

$$P(X = k) = \frac{\binom{K}{k}\binom{N-K}{n-k}}{\binom{N}{n}}$$

where:

- N is the population size.
- K is the number of success states in the population.
- n is the number of draws.
- k is the number of observed successes.

In R, you can simulate values from the hypergeometric distribution using the `rhyper` function available in the base package. The syntax for the `rhyper` function is `rhyper(nn, m, n, k)`, where `nn` is the number of observations, `m` is the population size, `n` is the number of items in the population with the desired characteristic, and `k` is the sample size.

For example, to simulate drawing 5 cards from a standard deck of 52 and seeing how many face cards (12 in a deck) you get, you can use:

```
# Simulating drawing 5 cards and checking the number of face cards
rhyper(10, 52, 12, 5)
#> [1] 4 3 4 4 5 3 4 3 4 5
```

To simulate a scenario where you draw 10 balls from an urn containing 50 balls of which 20 are red, and you're interested in the number of red balls you get:

```
# Simulating drawing 10 balls and checking the number of red balls
rhyper(10, 50, 20, 10)
#> [1] 7 6 7 6 5 8 8 7 8 9
```

2.5. Geometric Distribution. The **Geometric Distribution** emerges as a key concept in discrete distributions. It portrays the number of Bernoulli trials needed for a success to occur. This distribution is particularly relevant in scenarios where we're interested in identifying the likelihood of the first occurrence of a success, given a constant probability of success on each trial.[4]

[3]The hypergeometric distribution can be contrasted with the binomial distribution, which models the number of successes when drawing with replacement or when the probability of success remains constant in each trial.

[4]For context, if one were to examine the number of times one needs to flip a coin before getting heads, the geometric distribution would be the go-to model.

DEFINITION 5. A random variable X is said to have a **geometric distribution** with parameter p $(0 < p < 1)$ if its probability mass function is given by:

$$P(X = k) = (1 - p)^{k-1}p$$

where p is the probability of success on an individual trial and k is the trial number where the first success occurs.

In R, you can simulate values from the geometric distribution using the `rgeom` function available in the base package. The syntax for the `rgeom` function is `rgeom(n, prob)`, where `n` represents the number of observations and `prob` is the probability of success on any given trial.

For example, to simulate 10 trials and determine on which trial the first success occurs when the success probability is 0.3, you can use:

```
# Simulating 10 trials to see when the first success occurs
rgeom(10, 0.3)
#> [1] 4 1 2 1 5 3 6 1 3 1
```

Similarly, if you wish to see when the first success occurs in 15 trials, with a success probability of 0.5 (like flipping a coin):

```
# Simulating 15 trials for a fair coin flip until the first head appears
rgeom(15, 0.5)
#> [1] 2 1 1 3 1 1 2 2 1 1 3 2 1 1 2
```

2.6. Poisson Distribution. The **Poisson Distribution** stands as a pivotal model in the landscape of discrete probability distributions. It offers a way to represent the frequency of events within a defined period of time or space, assuming a consistent average rate of these events. Ideal for scenarios where occurrences are infrequent yet maintain a consistent average rate, the Poisson distribution proves invaluable. For instance, it can adeptly model the number of cars traversing a particular intersection over an hour or the volume of calls a call center manages within a designated timeframe.

The **Poisson Distribution** owes its name to the French mathematician Siméon Denis Poisson. He detailed it in his 1837 publication, *Recherches sur la probabilité des jugements en matière criminelle et en matière civile* [11]. This seminal work delved into the nuances of random variables and their ability to count distinct events over a specific duration. However, it's worth noting that the seeds of the discovery for this distribution trace back

to Abraham de Moivre in 1711. This earlier exploration has led some in the academic realm to argue that the credit for the distribution might be more aptly assigned to de Moivre [12].

DEFINITION 6. A random variable X is said to follow a **Poisson distribution** with parameter λ (where $\lambda > 0$) if its probability mass function is represented by:

$$P(X = k) = \frac{e^{-\lambda}\lambda^k}{k!}$$

where λ is the average rate of occurrence (expected number of occurrences in the interval), and k is the actual number of occurrences (a non-negative integer).

In R, the Poisson distribution can be simulated using the `rpois` function from the base package. The syntax for this function is `rpois(n, lambda)`, where `n` signifies the number of observations and λ is the expected number of occurrences in the interval.

For instance, to simulate the number of events occurring in 10 intervals, given an average rate of 3 events per interval, you would use:

```
# Simulating 10 intervals with an average rate of 3 events per interval
rpois(10, 3)
#> [1] 2 4 3 2 1 5 3 4 2 3
```

Similarly, if you want to understand the number of events in 15 intervals, where the average rate is 5:

```
# Simulating 15 intervals with an average rate of 5 events
rpois(15, 5)
#> [1] 4 6 5 5 4 6 7 4 5 4 6 5 5 4 6
```

2.7. Other Discrete Distributions. R provides several functions to generate random numbers from discrete distributions. Below are some of the distributions that are not covered in the module.

- **Negative Binomial Distribution** (`rnbinom(n, size, prob)`): Denotes the number of failures before the rth success. Parameters include `n` for observations, `size` for desired successes, and `prob` for success probability per trial.
- **Negative Hypergeometric Distribution** (`rnhyper(nn, m, n, k)`): Represents the number of failures before the kth success in sampling without replacement. Parameters

include `nn` for the number of observations, `m` for the population size, `n` for the number of success states in the population, and `k` for the number of successes in the sample.

- **Multinomial Distribution** (`rmultinom(n, size, prob)`): An extension of the binomial distribution with more than two results. Parameters are `n` for observations, `size` for trials, and `prob` as the probability vector for outcomes.

3. Continuous Distributions

3.1. Continuous Uniform Distribution. The continuous uniform distribution is defined as follows.

DEFINITION 7. A random variable X is said to have a **continuous uniform distribution** over the interval $[a, b]$ if its probability density function is given by

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

In R, the function `runif` is utilized to generate random numbers from a continuous uniform distribution. The numbers produced by this function are uniformly distributed between two given limits. If these limits are not explicitly provided, the default range is between 0 and 1. The basic syntax of the function is `runif(n, min = 0, max = 1)`, where `n` represents the number of random values you wish to generate, `min` signifies the lower boundary of the distribution, and `max` denotes its upper boundary. For example, to produce ten random numbers within the interval 1 to 3, one would invoke the command `runif(10, min=1, max=3)`.

```
# Generating ten continuous uniform distributed random numbers between 1 and
3
runif(10, min=1, max=3)
#> [1] 1.263228 1.210575 2.023167 1.600398 1.053434 1.619295 2.484239
1.070913 2.130152 1.560516
```

We will see later, the continuous uniform distribution in particularly useful when implementing the inverse CDF method.

3.2. Exponential Distribution. The exponential distribution is often used to model the time between the occurrence of events in a Poisson process, where events occur continuously and independently at a constant average rate.

DEFINITION 8. A random variable X is said to follow an **exponential distribution** with parameter λ (where $\lambda > 0$) if its probability density function is given by

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & \text{for } x \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

In R, the function `rexp` is used to generate random numbers from an exponential distribution. This function outputs numbers that decay at a rate specified by the user. The basic syntax for the function is `rexp(n, rate = λ)`, where `n` is the number of random values you want to generate, and `rate` is the rate parameter λof the distribution, if you don't input, then the default value it set to be 1 For instance, to obtain ten random numbers from an exponential distribution with a rate of 0.5, one would use the command `rexp(10, rate=0.5)`.

```
# Generating ten exponentially distributed random numbers with rate 0.5
rexp(10, rate=0.5)
#> [1] 0.4823 2.7328 1.9938 0.1554 0.8375 0.2251 3.6201 1.1830 0.8274 0.6932
```

The exponential distribution has various applications, including modeling the lifetime of certain products and the time between arrivals of a Poisson process.

3.3. Normal Distribution . The normal distribution, often referred to as the Gaussian distribution, is a continuous probability distribution characterized by a symmetrical bell-shaped curve. This curve is defined by two parameters: the mean (μ), which determines the center or the peak of the curve, and the standard deviation (σ), which denotes the spread or width of the curve. Its definition is as follows.

DEFINITION 9. A normal distribution is a type of continuous probability distribution for a real-valued random variable. The normal distribution is defined by its probability density function, given by:

$$\phi(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

where $\phi(x)$ is the probability density function, x is the random variable, μ is the mean or expectation of the distribution (which is also its median and mode), and σ is the standard deviation.

The probability density functions (PDFs) for normal distributions with various parameters are illustrated in Figure 2.

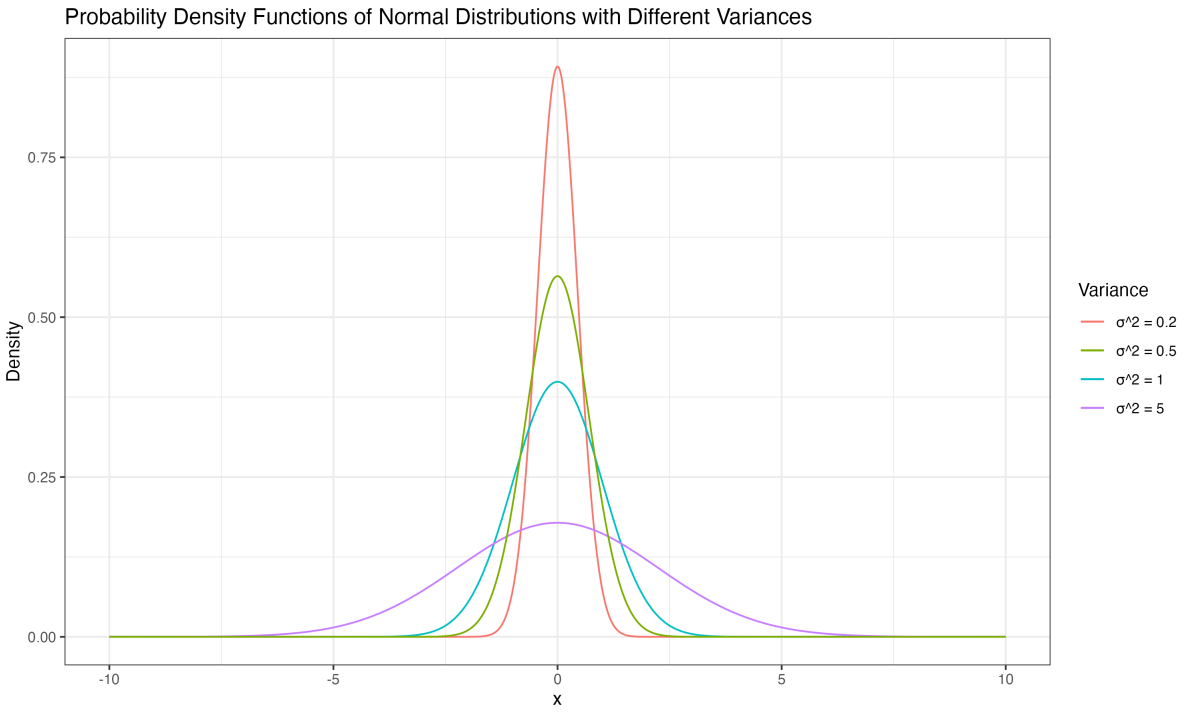


FIGURE 2. Probability density functions of various normal distributions (the blue curve is the standard normal distribution)

The **normal distribution** is also known as the Gaussian distribution, and it traces its origins back to the 18th and 19th centuries. Initially linked to Abraham de Moivre's work on the binomial distribution in 1738 [10]. However, it was Carl Friedrich Gauss who thoroughly described it in 1809 while rationalizing the method of least squares [7]. Gauss's contribution was so significant that the distribution is often attributed to him.

Figure 2 can be generated using the following R code.

```
# Load the ggplot2 package
library(ggplot2)

# Create a data frame to plot different normal distributions
x <- seq(-10, 10, by = 0.01)
df <- data.frame(
x = x,
y0.2 = dnorm(x, mean=0, sd=sqrt(0.2)),
y1 = dnorm(x, mean=0, sd=sqrt(1)),
y5 = dnorm(x, mean=0, sd=sqrt(5)),
y0.5 = dnorm(x, mean=0, sd=sqrt(0.5))
```

```
)

# Plot using ggplot2 with a minimal theme
p <- ggplot(df, aes(x = x)) +
geom_line(aes(y = y0.2, color = " ^2 = 0.2")) +
geom_line(aes(y = y1, color = " ^2 = 1")) +
geom_line(aes(y = y5, color = " ^2 = 5")) +
geom_line(aes(y = y0.5, color = " ^2 = 0.5")) +
labs(title = "Probability Density Functions of Normal
Distributions with Different Variances",
x = "x",
y = "Density",
color = "Variance") +
theme_minimal()

# Display the plot
print(p)

# Adjust the plot to use a black & white theme
p <- ggplot(df, aes(x = x)) +
geom_line(aes(y = y0.2, color = " ^2 = 0.2")) +
geom_line(aes(y = y1, color = " ^2 = 1")) +
geom_line(aes(y = y5, color = " ^2 = 5")) +
geom_line(aes(y = y0.5, color = " ^2 = 0.5")) +
labs(title = "Probability Density Functions of Normal
Distributions with Different Variances",
x = "x",
y = "Density",
color = "Variance") +
theme_bw()

# Save the plot as a PNG file
ggsave(filename = "normal_distributions.png", plot = p,
width = 10, height = 6, dpi = 300)
```

Here is the visualization is conducted using the `ggplot2` package, which is a popular data visualization library in R, based on the principles of "the Grammar of Graphics". It offers a

flexible and consistent system for creating a wide variety of static, interactive, and animated graphics. With its layered approach, users can incrementally add components to plots, making it both powerful and user-friendly for complex visualizations.

A fundamental concept closely associated with the normal distribution is the Central Limit Theorem (CLT), a fundamental concept in probability theory and statistics.

THEOREM 1. *Let $\{X_1, X_2, \ldots, X_n\}$ be a sequence of independent and identically distributed (i.i.d.) random variables with finite expected value $E[X_i] = \mu$ and finite variance $Var(X_i) = \sigma^2 > 0$. Define the sample mean as*

$$\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i.$$

Then, as $n \to \infty$, the distribution of the standardized sum

$$Z_n = \sqrt{n}\left(\bar{X}_n - \mu\right)$$

converges in distribution to a normal distribution, i.e.,

$$Z_n \xrightarrow{d} N(0, \sigma^2).$$

Note "$\xrightarrow{d}$" stands for "convergence in distribution", which is also known as weak convergence or convergence in law, is a fundamental concept in probability theory and statistics. It refers to the situation where a sequence of random variables (X_n) converges to another random variable X in terms of their cumulative distribution functions (CDFs). Hopefully, you will be familiar with after this course.

In other words, the average of a large number of i.i.d. random variables, $\bar{X}_n$, after appropriate standardization, will tend to follow a standard normal distribution, regardless of the original distribution of the individual variables. The Law of Large Numbers tells us that sample average $\bar{X}_n$ will converge almost surely to the expected value μ as n approaches infinity. The CLT goes further by describing the size and distributional form of the fluctuations around this deterministic number μ during this convergence. Specifically, it states that as n gets larger, the difference between the sample average $\bar{X}_n$ and its limit μ, when multiplied by $\sqrt{n}$, approximates a normal distribution with mean 0 and variance σ^2. This means that for large enough n, the distribution of $\bar{X}_n$ gets arbitrarily close to a normal distribution with mean μ and variance σ^2/n. This intrinsic property makes the normal distribution a pivotal tool in inferential statistics, as it ensures that statistical inferences derived from sample means are generally applicable.

Under certain mixing conditions, when the variables $\{X_1, \ldots, X_n\}$ are correlated, $\sqrt{n}(\bar{X}_n - \mu)$ also converges to a normal distribution. This notion is captured under various generalized versions of the CLT that account for dependent sequences. For correlated variables,

the mixing conditions become crucial. Mixing conditions are a set of criteria that, even in the presence of dependence, ensure a sort of "averaging out" effect over long ranges, so that the dependent structure does not overpower the tendency toward normality. While the intricacies of these mixing conditions and the convergence of correlated variables to a normal distribution are pivotal for advanced statistical understanding, they are beyond the scope of this module. As you progress in your mathematical journey and mature into a mathematics graduate, these concepts will become more comprehensible.

To generate normally distributed random variables is straightforward.

```
# Generating ten standard normally distributed random numbers
rnorm(10, 0, 1)
#> [1] -0.8267256 -0.4518683 -1.1265701 -1.7666592 -1.2771818 -1.1659479
1.3600173 -0.5853499 0.8315215 1.3784917
```

We could simply put down `rnorm(10)` if we were only interested in standard normal random numbers, because the function `rnorm` sets the mean and standard deviation to be 0 and 1, respectively, by default.

```
rnorm(10)
#> [1] 1.574904176 -0.572594806 0.008159591 -0.472049254 0.342730428
-0.194104579 -0.080820534 0.247510159 1.122421265 2.135382611
```

If we are interested in generating 5 normally distributed random variables with a mean of 1 and a standard deviation of 2, then we should change the code as follows.

```
rnorm(5, 1, 2)
#> [1] 0.9065939 1.6503606 1.3322302 1.3874926 -0.4869327
```

Note that due to the pseudo-random nature of the results, you may see different output.

Practicing using R while learning the language is essential, so here is an exercise for you.

Exercise: If I would like to generate 100 random standard normal variables with a mean of 2 and a variance of 10, what should I do?
Hint: The inputs of `rnorm` are the number of random variables, mean, and **standard deviation**.

3.4. Other Continuous Distributions. R offers a variety of functions to generate random numbers from other continuous distributions as well.

- **Weibull Distribution** (`rweibull(n, shape, scale)`): Simulates random numbers from a Weibull distribution. Here, `shape` and `scale` are the shape and scale parameters, respectively.
- **Chi-Square Distribution** (`rchisq(n, df)`): Produces random numbers from a chi-square distribution with `df` degrees of freedom.
- **Student's t Distribution** (`rt(n, df)`): Generates random numbers from a Student's t-distribution with `df` degrees of freedom.
- **F Distribution** (`rf(n, df1, df2)`): Simulates random numbers from an F distribution. `df1` and `df2` are the degrees of freedom for the numerator and the denominator, respectively.

Exercises: You are encouraged to experiment with these commands in R to familiarize yourself with their properties and applications.

4. More Discussion on Random Variable Generators

So far, we have only used these random generators to generate random variables, but we can also use related functions to generate CDF, quantile, etc. Let's consider the normal distribution. Recall that if we forget the syntax of a command, e.g. `rnorm`, we could always type the command into the "Help" tab, which allows us to search the R documentation for assistance, as shown in Figure 3.

Following this procedure will direct us to the webpage: `https://stat.ethz.ch/R-manual/R-devel/library/stats/html/Normal.html`. The content of the webpage is partially excerpted as follows:

Description

Density, distribution function, quantile function and random generation for the normal distribution with mean equal to mean and standard deviation equal to sd.

Usage

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

Arguments

`x, q`: vector of quantiles.
`p`: vector of probabilities.
`n`: number of observations. If `length(n) > 1`, the length is taken to be the number required.
`mean`: vector of means.
`sd`: vector of standard deviations.
`log, log.p`: logical; if TRUE, probabilities p are given as log(p).
`lower.tail`: logical; if TRUE (default), probabilities are $P[X \leq x]$ otherwise, $P[X > x]$.

Details

If mean or sd are not specified they assume the default values of 0 and 1, respectively.
The normal distribution has density:

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

where μ is the mean of the distribution and σ the standard deviation.

Value

`dnorm` gives the density, `pnorm` gives the distribution function, `qnorm` gives the quantile function, and `rnorm` generates random deviates. ...

As you can see from Figure 3 and the help page (also known as the documentation page) at `https://stat.ethz.ch/R-manual/R-devel/library/stats/html/Normal.html`, the help page provides detailed information on the normal distribution functions in the R programming language. It elaborates on functions such as `dnorm`, `pnorm`, `qnorm`, and `rnorm`, along with their respective parameters and options.

- `dnorm`: Computes the density of the normal distribution for a given set of values.
- `pnorm`: Gives the cumulative distribution function, which is the probability that a normally distributed random number will be less than a given value.
- `qnorm`: Returns the quantile function, indicating the value below which a given percentage of observations in a group fall.
- `rnorm`: Generates random numbers from a normal distribution with a specified mean and standard deviation.

It also includes references and example implementations.

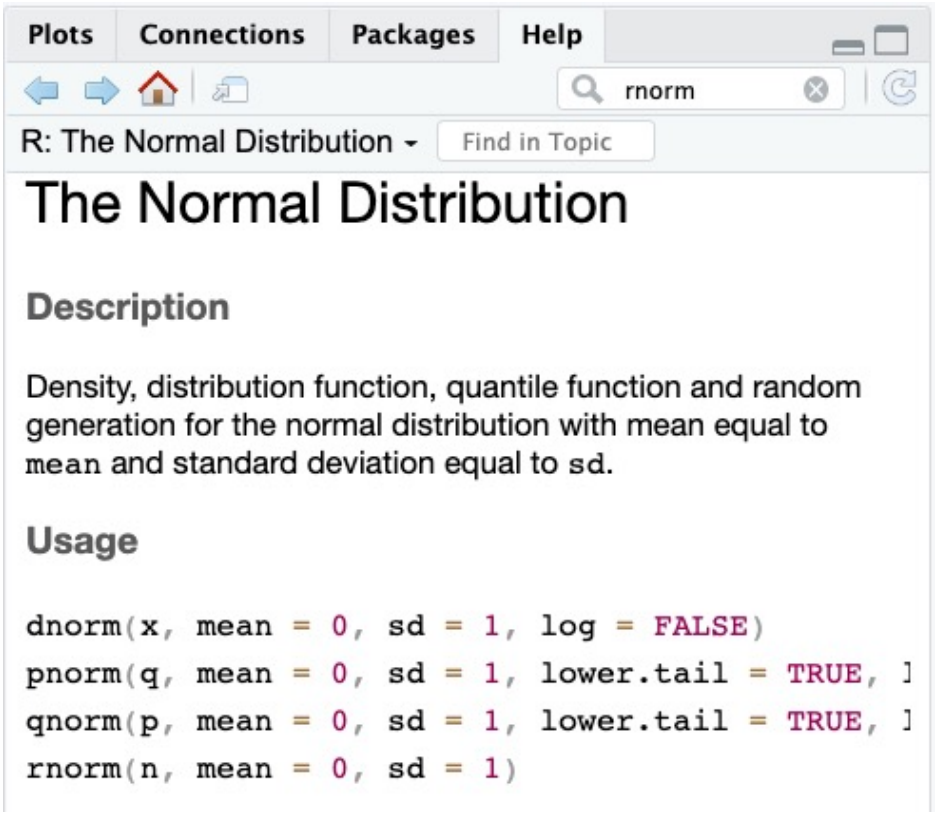


FIGURE 3. The "Help" tab

Exercise: Find the critical values for all the distributions discussed so far. Use a 5% significance level. For two-tailed distributions, provide values for both tails. For distributions defined only for positive values, provide the one-tailed value.

5. The inverse CDF method

Please note that the inverse CDF method is not a requirement for this course; it is included here merely for informational purposes. As you delve deeper into data science, you may encounter nonstandard distributions for which R does not have a built-in random generator. In such cases, you'll need to code the random variable generation yourself.

The inverse CDF method, also known as the quantile transformation method, is used to generate random variables with a specified distribution. The main idea behind this method is as follows:

Let U be a random variable uniformly distributed on the interval $(0, 1)$. For a continuous random variable X with cumulative distribution function (CDF) $F_X(x)$, the transformation

$$X = F_X^{-1}(U)$$

will result in a random variable with CDF $F_X(x)$.

The steps to implement the inverse CDF method are:

(1) Generate a random number u uniformly distributed in $(0, 1)$.
(2) Compute $x = F_X^{-1}(u)$, where $F_X^{-1}(u)$ is the inverse of the CDF of the desired distribution.
(3) The value of x is a realization of the random variable with distribution $F_X(x)$.

To successfully employ the inverse CDF method, it's crucial to have an analytical form of the inverse CDF $F_X^{-1}(u)$ for the desired distribution.

Let's consider a toy example. Suppose we don't have the `rnorm` function. Instead, we only possess the `runif` function, and our aim is to generate a series of random variables that follow a standard normal distribution. We've chosen this particular example because the simulated results can be easily verified, ensuring you can confirm the validity of the algorithm.

The code below uses the inverse CDF method to generate a series of random variables that follows the standard normal distribution.

```
rm(list=ls())
set.seed(123456789 )
######################################################################
# CDF of a standard normal distribution
compute_phi <- function(z) {
integrand <- function(t) {
(1/sqrt(2 * pi)) * exp(-t^2 / 2)
}
result <- integrate(integrand, -Inf, z)
return(result$value)
}

# Test the function
z_val <- 1.96
print(compute_phi(z_val))

# Compare with built-in function
print(pnorm(z_val))

# Use the inverse CDF method.

# First, sample from the continuous uniform distribution on [0,1]
sample.unif <- runif(1000)

# The inverse of CDF for the standard normal distribution is
inverse_phi <- function(phi_value) {
```

```
function_to_zero <- function(z) {
compute_phi(z) - phi_value
}
root <- uniroot(function_to_zero, c(-10, 10))
return(root$root)
}

# Sample from the standard normal distribution
sample.norm <- as.numeric(lapply(sample.unif, inverse_phi))

# Plot histogram
hist(sample.norm, probability = TRUE, main = "Histogram with Density
Curve",  xlab = "Value", col = "lightblue", border = "black")

# Overlay empirical density curve
lines(density(sample.norm), col = "red", lwd = 2)
```

Below is a breakdown of this code, and what it does.

(1) **Initialization**:
   - `rm(list=ls())`: Removes all variables currently present in the R environment to start fresh.
   - `set.seed(123456789)`: Sets the seed for random number generation in R. This ensures that any random processes will produce the same results every time this script is run.

(2) **Defining the Cumulative Density Function (CDF) of a Standard Normal Distribution**:
   - The function `compute_phi(z)` computes the CDF of a standard normal distribution up to a given value `z`. Note that the CDF of a standard normal distribution is
$$\Phi(z) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{z} e^{-\frac{t^2}{2}}\, dt.$$
   - Note when computing the CDF, we use `integrate` to numerically integrates the `integrand` from `-Inf` up to the value `z`.

(3) **Testing the `compute_phi` function**:
   - The function is tested using a `z` value of `1.96`.
   - The result is compared with R's built-in `pnorm` function for validation. They do match up!

(4) **Sampling using Inverse CDF Method – Step 1**:

- Generates a sample from a continuous uniform distribution between 0 and 1 using `runif`.

(5) **Inverse of the CDF**:
- The function `inverse_phi` finds the inverse of the CDF for the standard normal distribution.
- The goal of `function_to_zero` is to find a `z` value where the difference between the calculated CDF and a given `phi_value` is zero.
- `uniroot` finds the root of `function_to_zero` within a specified interval.

(6) **Sampling using Inverse CDF Method – Step 2**:
- For each value in `sample.unif`, the corresponding `z` value in a standard normal distribution is found using the `inverse_phi` function.

(7) **Plotting the histogram and empirical density**:
- The `hist` function plots a histogram of the samples.
- The `lines` function overlays an empirical density curve on the histogram.

The histogram and the empirical density curve is plotted as follows.

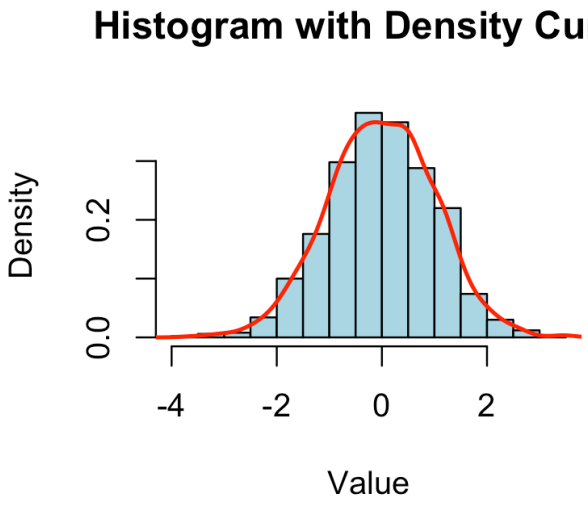


FIGURE 4. Histogram with Density Curve

The empirical density curve is a non-parametric representation of the distribution of a dataset. It is constructed by smoothing the histogram of the data, thereby providing a visual depiction of its distribution without making any assumptions about the specific form of the underlying distribution. The concept falls within the domain of nonparametric statistics, which is beyond the scope of this course. However, it's noteworthy to mention that our simulated results from the inverse CDF algorithm align graphically with the standard normal distribution when visualized using the empirical density curve.

Furthermore, a histogram can be viewed as a nonparametric estimator for the density of data, when a uniform kernel is applied. In essence, the histogram divides the data into intervals (or bins) and counts the number or proportion of data points in each interval, thereby estimating the underlying distribution without making explicit assumptions about its form. This perspective, seeing the histogram as a nonparametric density estimator with a uniform kernel, delves into deeper statistical concepts and is also beyond the scope of this course.

If the visualization doesn't convince you, then perhaps we could also examine the statistical properties of the simulated data, e.g. its mean and variance.

```
> # Examine the statistical properties of the generated data
> mean(sample.norm
[1] -0.007377702
> var(sample.norm)
[1] 1.0353
```

The results are pretty close to 0 and 1, the theoretical values! Because of the randomness involved, and the fact we can't have a sample size of infinity, there always going to be slight deviation from the theoretical values.

CHAPTER 4

Conditional Probability and Independence

In this chapter, we delve into the intricate world of probability concepts, leveraging the computational capabilities of R for a hands-on and insightful exploration. We will begin by exploring the essence of **conditional probability**, which delves into the probability of an event A occurring when another event B has already transpired. Using R, practical scenarios such as the probability of rolling an even number given that the roll produced a value greater than three can be visualized. As we progress, the focus will shift to computing *conditional probabilities* in basic scenarios, guided by formulas such as $P(A|B) = \frac{P(A \cap B)}{P(B)}$. Subsequently, the **Law of Total Probability** and **Bayes' Formula** will be introduced as tools that allow for a broader understanding of interconnected events, with dice simulations breathing life into these abstract concepts. Central to our discussion will be the principle of *independence*, where we will investigate how events can remain uninfluenced by each other's outcomes. Through R simulations, the chapter should offer a deep dive into verifying the independence of events and using this knowledge to simplify probability calculations.

1. Monte Carlo Simulations

Because these concepts' validation depends on approximations that become more precise as the sample size grows, we frequently rely on what are termed "Monte Carlo" simulations. The Monte Carlo simulation is a mathematical technique devised to comprehend the uncertainty and variability in models. Named after the renowned Monaco casino, it underscores its dependency on randomness and chance. At its core, the method involves repeatedly executing a random data-generating process to observe a spectrum of possible outcomes. For illustration, consider simulating a dice roll thousands of times to discern the most recurring results. Within statistics, Monte Carlo simulations furnish a method for prognostication and decision-making amidst a myriad of potential scenarios. This provides students with a tangible tool to unravel complex issues in a more accessible format.

It's noteworthy that these Monte Carlo simulations are intricately tied to the behavior of the average of a series of random variables as the sample size expands. This behavior correlates with the so-called limit theorems in statistics, which encompass the Central Limit Theorem and the Law of Large Numbers. I've intentionally used the full terminology, eschewing abbreviations, to underscore the significance of these concepts. The Central Limit Theorem was elaborated upon in Section 3.3, and the Law of Large Numbers is delineated below.

THEOREM 2 (Strong Law of Large Numbers). *Let $X_1, X_2, \ldots$ be a sequence of independent and identically distributed (i.i.d.) random variables with finite expected value μ. Then, for almost every sample path,*

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} X_i = \mu \quad a.s.$$

This theorem states that the sample average converges almost surely, abbreviated as "a.s.", to the expected value μ as the sample size grows to infinity. Note "almost surely" is a technical term in probability theory. When we say an event happens "almost surely", we mean that the event occurs with probability 1.

Here we demonstrate the idea of Monte Carlo simulations through an example, namely, how to estimate the number π.

Randomly place points in a square. If we inscribe a circle inside this square and randomly scatter points, the ratio of the number of points inside the circle to the total number of points should approximate the ratio of the areas of the circle and the square. Given the area of the square, we can then estimate π.

Steps:

(1) For a circle of radius r inscribed in a square with side length $2r$, the area of the circle is πr^2 and the area of the square is $4r^2$.
(2) The ratio of the areas is $\frac{\pi r^2}{4r^2} = \frac{\pi}{4}$.
(3) By randomly placing points in the square, the ratio of points inside the circle to the total number of points in the square should approximate $\frac{\pi}{4}$. Therefore, π can be approximated as $4\times$ this ratio.

To implement this in R:

```
set.seed(1234567)  # Setting a seed for reproducibility

n_points <- 10000  # Number of random points

# Generate random points in the square [-1, 1] x [-1, 1]
x <- runif(n_points, min=-1, max=1)
y <- runif(n_points, min=-1, max=1)

# Determine how many points fall inside the circle of radius 1
inside_circle <- sum(x^2 + y^2 <= 1)

# Estimate pi
```

```
pi_estimate <- 4 * inside_circle / n_points

print(pi_estimate)
```

Explanation of the R code for estimating π is as follows.

(1) `set.seed(1234567)`:
- Initializes the random number generator in R.
- Provides consistent and reproducible sequences of random numbers every time the script is run by setting a specific seed value of 1234567.

(2) `n_points <- 10000`:
- Creates a variable `n_points` and assigns the value 10,000 to it.
- Represents the total number of random points for the simulation.

(3) `x <- runif(n_points, min=-1, max=1)` and `y <- runif(n_points, min=-1, max=1)`:
- Generates the x and y coordinates of the 10,000 random points.
- Uses the `runif()` function to generate uniformly distributed random numbers between -1 and 1 for both x and y coordinates. Represents points within the square [-1, 1] x [-1, 1].

(4) `inside_circle <- sum(x^2 + y^2 <= 1)`:
- Determines the number of random points that fall inside a circle of radius 1 centered at the origin (0,0).
- Checks each point's location based on the circle's equation. The `sum()` function counts the number of points inside the circle.

(5) `pi_estimate <- 4 * inside_circle / n_points`:
- Estimates the value of π.
- Uses the ratio of points inside the circle to total points to approximate $\frac{\pi}{4}$. Multiplying by 4 gives the estimate for π.

(6) `print(pi_estimate)`:
- Prints the estimated value of π to the console.

The essence of this code is to use the Monte Carlo method to approximate the value of π. It generates random points within a square and determines how many of those points lie within an inscribed circle.

The output will provide an approximation of π. Increasing the value of `n_points` will improve the accuracy of the approximation.

Exercises: Can you think of some more scenarios where Monte Carlo simulations would be useful?

2. Conditional Probability

```
# Simulate 10000000 rolls of a dice
set.seed(1234567)
rolls <- sample(1:6, 10000000, replace=TRUE)

# P(A): Probability that the roll is even
P_A <- mean(rolls %% 2 == 0)

# P(B): Probability that the roll is greater than 3
P_B <- mean(rolls > 3)

# P(A|B): Probability that the roll is even given it's greater than 3
P_A_given_B <- mean(rolls[rolls > 3] %% 2 == 0)
P_A_given_B
```

This R code demonstrates the concept of conditional probability using simulated dice rolls:

(1) **Setting the Random Seed**:
`set.seed(1234567)`
This function ensures reproducibility of random operations. With this seed, any random process will produce the same results every time it's run.

(2) **Simulating Dice Rolls**:
`rolls <- sample(1:6, 10000000 replace=TRUE)`
This line simulates 10000000 rolls of a fair six-sided dice. The `sample` function randomly selects numbers from 1 to 6 (inclusive), repeating this process 1000 times with replacement.

(3) **Probability of an Even Roll (P(A))**:
`P_A <- mean(rolls %% 2 == 0)`
This computes the probability of an even roll. The modulo operator (`%%`) returns the remainder after division. The subsequent comparison checks if the roll is even, and `mean` calculates the proportion, which is the probability $P(A)$.

(4) **Probability of a Roll Greater than 3 (P(B))**:
`P_B <- mean(rolls > 3)`
This calculates the probability $P(B)$ of obtaining a roll greater than 3. It checks which rolls exceed 3 and then computes the proportion using `mean`.

(5) **Conditional Probability (P(A|B))**:
`P_A_given_B <- mean(rolls[rolls > 3] %% 2 == 0)`
Here, the conditional probability $P(A|B)$ is computed, representing the likelihood of an even roll when the result is greater than 3. The subset `rolls[rolls > 3]` focuses on

rolls exceeding 3. For these, the code checks evenness and then computes the conditional probability using `mean`.

(6) **Displaying the Result**:

```
P_A_given_B
```

This line simply displays the computed $P(A|B)$.

This R code enables the simulation of 1000 dice rolls to calculate and illustrate various probability concepts, notably the conditional probability of obtaining an even result given that the dice roll exceeds 3.

3. Computing Conditional Probabilities in Simple Scenarios

```
# P(A and B): Probability that roll is even and greater than 3
P_A_and_B <- mean(rolls > 3 & rolls %% 2 == 0)

# Compute P(A|B) using the formula
P_A_given_B_formula <- P_A_and_B / P_B
P_A_given_B_formula
```

The R code segment presented explores joint and conditional probabilities:

(1) **Joint Probability $P(A$ and $B)$**:

```
P_A_and_B <- mean(rolls > 3 & rolls %% 2 == 0)
```

This line calculates the joint probability $P(A$ and $B)$ that a dice roll is both even (A) and greater than 3 (B). The condition `rolls > 3` checks for numbers greater than 3 while `rolls %% 2 == 0` ensures they are even. The conjunction `&` ensures both conditions are met, and `mean` computes the proportion of rolls that satisfy both, which represents the joint probability.

(2) **Computing Conditional Probability using the Formula**:

```
P_A_given_B_formula <- P_A_and_B / P_B
```

This line calculates the conditional probability $P(A|B)$ using the formula:

$$P(A|B) = \frac{P(A \text{ and } B)}{P(B)}$$

The code divides the previously computed joint probability $P(A$ and $B)$ by the probability $P(B)$, effectively deriving the conditional probability $P(A|B)$ which indicates the likelihood of a roll being even when it's known to be greater than 3.

(3) **Displaying the Result**:

```
P_A_given_B_formula
```

This command simply outputs the computed conditional probability $P(A|B)$ based on the formula.

In summary, this R code segment focuses on the interconnectedness of events, particularly the scenario where a dice roll is even and exceeds 3. The calculations and results provide insights into the foundational probability concepts of joint and conditional probabilities.

4. The Law of Total Probability and Bayes' Formula

```
# Consider two dice rolls: A is the event both dice sum to 7.
# B1 is the event the first dice is 1, B2 is 2, etc.
dice <- expand.grid(1:6, 1:6)
P_A_given_B <- numeric(6)
P_B <- 1/6

for(i in 1:6) {
P_A_given_B[i] <- mean(dice$Var1 + dice$Var2 == 7 & dice$Var1 == i)
}

# Law of total probability
P_A <- sum(P_A_given_B * P_B)

# Assuming we know event A occurred,
# Bayes' formula to find probability the first dice was a 2:
P_B2_given_A <- (P_A_given_B[2] * P_B) / P_A
```

This R code segment illustrates concepts related to conditional probability, the law of total probability, and Bayes' theorem, all using the example of two dice rolls:

(1) **Setting Up the Scenario**:

```
dice <- expand.grid(1:6, 1:6)
```

The `expand.grid` function creates a data frame where each row represents a possible outcome of rolling two six-sided dice. For instance, a row with values 2 and 3 represents the event that the first dice showed a 2 and the second dice showed a 3.

(2) **Initialization**:

```
P_A_given_B <- numeric(6)
P_B <- 1/6
```

Here, an empty numeric vector of length 6 is initialized to store the conditional probabilities $P(A|B_i)$ for each possible value of the first dice. The probability $P(B)$ for any specific outcome of the first dice (like rolling a 1, 2, etc.) is $\frac{1}{6}$ since it's a fair dice.

(3) **Computing Conditional Probabilities**:

```
for(i in 1:6) {
P_A_given_B[i] <- mean(dice$Var1 + dice$Var2 == 7 & dice$Var1 == i)
```

```
}
```

This loop computes the conditional probabilities $P(A|B_i)$ for $i = 1$ to 6. For each i, it calculates the probability that the sum of the two dice is 7 given that the first dice showed a value of i.

(4) **Law of Total Probability**:

```
P_A <- sum(P_A_given_B * P_B)
```

The probability $P(A)$ is calculated using the law of total probability. This expresses the probability of A (both dice summing to 7) by weighting each $P(A|B_i)$ with the probability $P(B)$ and summing them up.

(5) **Applying Bayes' Theorem**:

```
P_B2_given_A <- (P_A_given_B[2] * P_B) / P_A
```

Given the occurrence of event A (both dice summing to 7), this line uses Bayes' theorem to compute the probability $P(B_2|A)$, which represents the likelihood that the first dice showed a 2.

Overall, this code provides a systematic approach to understand the interplay between conditional probabilities, the law of total probability, and Bayes' theorem in the context of dice rolls.

5. Understanding the Concept of Independence

```
# With dice rolls, the roll of one dice doesn't
# affect the other, so they're independent

# P(A): Probability that the roll is even
P_A <- mean(rolls %% 2 == 0)

# P(B): Probability that the roll is greater than 3
P_B <- mean(rolls > 3)

# P(A and B): Probability that roll is even and greater than 3
P_A_and_B <- mean(rolls > 3 & rolls %% 2 == 0)

tolerance <- 1e-1
independent <- abs(P_A_and_B - (P_A * P_B)) < tolerance
independent
```

The R code segment in question aims to determine the independence of two events. For two events to be deemed independent, the probability of both events occurring together, $P(A \text{ and } B)$, should be equal to the product of their individual probabilities, $P(A) \times P(B)$. The `all.equal`

function in R is used to check if these two quantities are effectively the same (to a certain numerical precision).

If the `independent` variable is `TRUE`, then the events are considered to be independent. If it's not `TRUE` (i.e., if it's `FALSE`), then the two events are not independent. As the real-world computational results might differ slightly due to floating-point arithmetic, but they should be close enough to be considered equal for practical purposes. Thus, we set a threshold of tolerance, i.e. 0.1), if the absolute difference between the left and right sides of this equation is smaller than 0.1), then the events A and B can be considered independent based on our simulated results.

Note: It is essential to redefine the probabilities $P(A)$ and $P(B)$ in the segment of the code to guarantee the usage of accurate probabilities. This refinement is necessary because we set $P(B) = 1/6$ when illustrating the law of total probability and Bayes' formula earlier.

6. Using Independence to Simplify Probability Calculations

```
# Joint probability of rolling a 3  # on the first dice
# and a 4 on the second dice
P_roll3 <- mean(dice$Var1 == 3)
P_roll4 <- mean(dice$Var2 == 4)

joint_prob <- P_roll3 * P_roll4
```

The provided R code segment calculates the joint probability of two independent events regarding dice rolls:

- **Probability of rolling a 3 on the first dice (A)**:
  `P_roll3 <- mean(dice$Var1 == 3)`
  This computes the likelihood $P(A)$ that the first dice results in a 3.
- **Probability of rolling a 4 on the second dice (B)**:
  `P_roll4 <- mean(dice$Var2 == 4)`
  This calculates the likelihood $P(B)$ that the second dice shows a 4.
- **Joint Probability of A and B**:
  `joint_prob <- P_roll3 * P_roll4`
  Given the independence of the dice rolls, the joint probability $P(A \text{ and } B)$ is the product of $P(A)$ and $P(B)$.

Essentially, the code determines the chance of rolling a 3 on the first dice and a 4 on the second dice simultaneously.

CHAPTER 5

Limit Theorems

In this chapter, we focus on limit theorems, such as the Law of Large Numbers and the Central Limit Theorem, which are covered in Sections 3.3 and 1. Note that I deliberately avoided using abbreviations because I wanted to underscore their significance in probability and statistics. We will explore their properties using R codes and simulations, hoping to provide you with valuable statistical and probabilistic insights that will benefit your studies.

1. Law of Large Numbers

Here, we demonstrate the Law of Large Numbers through using coin flips:

```
flips <- sample(c(0, 1), 1000, replace = TRUE)
cum_avg <- cumsum(flips) / (1:1000)
plot(cum_avg, type = "l", ylim = c(0, 1), ylab = "Cumulative Average",
         xlab = "Number of Flips")
abline(h=0.5, col="red")
```

Clearly, from Figure 1, the average converges to 0.5, as the number of flips (sample size) increases.

The R code above carries out the following actions.

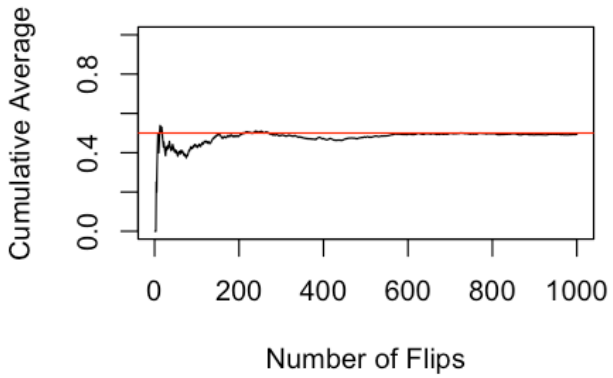


FIGURE 1. Demonstration of the Law of Large Number using coin flips.

(1) A sequence of 1000 coin flips is simulated with `sample(c(0, 1), 1000, replace = TRUE)`, where a result of 0 might signify "tails" and 1 "heads".
(2) The cumulative average of the coin flips, representing the proportion of "heads" encountered up to each flip, is calculated with `cumsum(flips) / (1:1000)`.
(3) A line graph is then plotted showing the cumulative average of "heads" against the number of flips made. The y-axis has a range between 0 and 1 to accommodate the range of possible averages for the flips.
(4) Lastly, a horizontal red line is drawn on the plot at the y-value of 0.5, symbolizing the expected average for a fair coin toss, where there's an equal probability for "heads" or "tails".

2. Central Limit Theorem

Here we use a Monte Carlo simulation to demonstrate the Central Limit Theorem .

First, we consider the De Moivre-Laplace Theorem, which is a precursor to the Central Limit Theorem for binomial distributions. For a binomial random variable X with parameters n and p, as n tends to infinity (with p fixed), the distribution of the standardized variable

$$Z = \frac{X - np}{\sqrt{np(1-p)}}$$

approaches the standard normal distribution.

We demonstrate the validity of De Moivre-Laplace Theorem as follows:

```
n <- 1000
p <- 0.5
binom_samples <- rbinom(10000, size=n, prob=p)
normalized_samples <- (binom_samples - n*p) / sqrt(n*p*(1-p))
hist(normalized_samples, breaks=50, prob=TRUE,
main="De Moivre-Laplace Demonstration", xlab="Value")
curve(dnorm(x), add=TRUE, col="red")
```

The R code can be explained as follows:

(1) The parameters n and p are set to 1000 and 0.5, respectively. Here, n denotes the number of Bernoulli trials, while p signifies the success probability for each trial.
(2) Next, 10,000 samples are drawn from a binomial distribution with parameters n and p using `rbinom`. Each sample represents the count of successes in 1000 trials.
(3) These binomial samples are then standardized according to the De Moivre-Laplace theorem. This normalization transforms the binomial distribution to approximate a standard normal distribution.

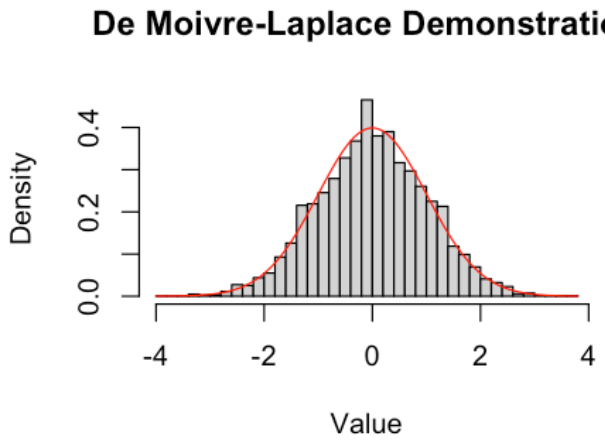


FIGURE 2. Demonstrate the De Moivre-Laplace Theorem using R.

(4) A histogram of the normalized samples is plotted with 50 bins, showcasing the distribution of the data. The histogram's title is set as "De Moivre-Laplace Demonstration", and the x-axis is labeled "Value".
(5) To emphasize the approximation to the normal distribution, a standard normal curve (mean = 0 and standard deviation = 1) is overlaid on the histogram in red.

Clearly, from Figure 2, the De Moivre-Laplace Theorem is valid.

Recall that we defined the Central Limit Theorem in Theorem 1. We use dice rolls, which follow a discrete uniform distribution. In this simulation, we roll a dice n_{dice} times and then average the results. We repeat this n_{rolls} times to produce a histogram of sample means. The Central Limit Theorem predicts that, for a sufficiently large n_{dice}, this histogram will approximate a normal distribution with mean $\mu = 3.5$ (the expected value of a dice roll) and standard deviation

$$\sigma = \sqrt{\frac{35}{12n_{\text{dice}}}}$$

(given the variance of a dice roll is $\frac{35}{12}$ and we're considering the distribution of the average).

```
# Monte Carlo Simulation of the Central Limit Theorem

# Number of dice
n_dice <- 10

# Number of rolls
n_rolls <- 10000

# Simulation
```

```
simulated_sums <- replicate(n_rolls,
sum(sample(1:6, n_dice, replace=TRUE)))

# Normalize the sums to get sample means
simulated_means <- simulated_sums / n_dice

# Plot the histogram of the sample means
hist(simulated_means, breaks=50, probability=TRUE,
main="Demonstration of Central Limit Theorem ",
xlab="Sample Mean of Dice Rolls",
ylab="Density",
col="lightblue", border="black")

# Add a density curve of a normal distribution for comparison
x_vals <- seq(1, 6, length=100)
norm_density <- dnorm(x_vals, mean=3.5, sd=sqrt(35/12)/sqrt(n_dice))
lines(x_vals, norm_density, col="red", lwd=2)

legend("topright", legend=c("Simulated Data", "Normal Distribution"),
lty=c(1,1), col=c("black", "red"))
```

The step-by-step explanation of the R code above is as follows:

(1) The variable `n_dice` is set to 10, which represents the number of dice that will be rolled in each trial. The variable `n_rolls` is set to 10,000, representing the total number of trials (or repetitions) for the simulation.
(2) Using the `replicate` function, the code performs 10,000 trials. In each trial, 10 dice are rolled (values between 1 and 6) and their outcomes are summed. This results in a set of 10,000 sums.
(3) The sums from the previous step are then normalized by dividing by `n_dice`. This produces the average (or mean) dice roll for each trial, resulting in a distribution of sample means. This is stored in the variable `simulated_means`.
(4) A histogram of the `simulated_means` is plotted to visualize the distribution. The histogram showcases the distribution of the sample means with 50 bins. It is labeled "Demonstration of Central Limit Theorem " and has axes labels for clarity.
(5) To emphasize the Central Limit Theorem 's prediction, a normal distribution curve with a mean of 3.5 (the expected value for a dice roll) and a standard deviation calculated from

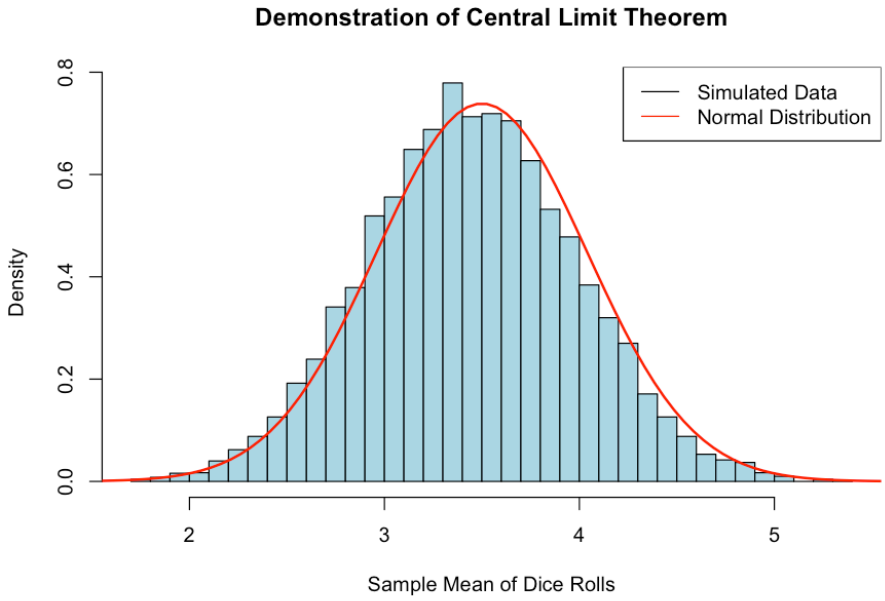


FIGURE 3. Monte Carlo Simulation of the Central Limit Theorem - Using Dice Rolls.

the variance of a dice roll and the sample size is overlaid on the histogram in red. This curve is expected to fit closely with the histogram as `n_dice` increases.

(6) Lastly, a legend is added to the top right corner of the plot to distinguish between the simulated data (histogram) and the expected normal distribution (red curve).

Clearly, Figure 3 indicates the validity of the Central Limit Theorem .

3. Markov's Inequality

Markov's Inequality provides an upper bound on the probability that a non-negative random variable takes on values that are a certain number of times larger than its expected value. Below is how that is demonstrated in R.

```
samples <- abs(rnorm(1000))
a <- 3
empirical_prob <- mean(samples >= a)
markov_bound <- mean(samples) / a
empirical_prob <= markov_bound
```

The R code above performs the following operations:

(1) It generates 1000 random numbers from a standard normal distribution and converts them to their absolute values, ensuring they are all positive. This collection of numbers is stored in the variable `samples`.

(2) A threshold value of 3 is assigned to the variable a.
(3) The empirical probability of the numbers in `samples` being greater than or equal to a is computed.
(4) Markov's inequality bound is calculated for the data. This bound states that for any non-negative random variable X and $a > 0$, the probability that X is at least a is bounded above by the expected value of X divided by a.
(5) The code concludes by comparing the empirical probability with the Markov's inequality bound to check if the former is less than or equal to the latter.

4. Chebyshev's Inequality

```
samples <- rnorm(1000, mean=5, sd=2)
k <- 2
empirical_prob <- mean(abs(samples - mean(samples)) >= k*sd(samples))
chebyshev_bound <- 1 / (k^2)
empirical_prob <= chebyshev_bound
```

The R code can be understood as follows:

(1) It generates 1000 random numbers from a normal distribution with a mean of 5 and a standard deviation of 2. These are stored in the variable `samples`.
(2) A constant factor of 2 is assigned to the variable k.
(3) The empirical probability is calculated, which represents the proportion of numbers in `samples` that deviate from the sample's mean by at least k times the sample's standard deviation.
(4) Using Chebyshev's inequality, a bound is calculated. The inequality asserts that for any random variable X with finite expected value and variance, the probability that X deviates from its mean by at least k times its standard deviation is at most $\frac{1}{k^2}$.
(5) Lastly, a comparison is made to check if the empirical probability is less than or equal to the Chebyshev bound.

Conclusion

This book/handout provides a comprehensive overview of R programming, set against the backdrop of statistics and probability. It methodically traces the journey from the historical origins of R, through its fundamental programming aspects, and dives into advanced statistical concepts. The book's structure, starting with the basics like data structures and advancing to profound topics like the Central Limit Theorem and Bayes' Formula, ensures clarity and progression for its readers. The emphasis on real-world application underlines the relevance of R in contemporary data analysis. By encompassing both foundational and advanced statistical concepts within the context of R, this handout presents itself as a balanced resource, well-suited for those eager to navigate the intricacies of data in the modern world.

Bibliography

[1] Richard A Becker and John M Chambers. *S: an interactive environment for data analysis and graphics*. CRC Press, 1984.

[2] Richard A Becker and John M Chambers. *Extending the S Systems*. Pacific Grove, CA, USA: Wadsworth & Brooks Cole, 1985.

[3] John M Chambers. *Programming with data: A guide to the S language*. Springer Science & Business Media, 1998.

[4] John M Chambers and Trevor J Hastie. Statistical models. In *Statistical models in S*, pages 13–44. Routledge, 2017.

[5] Peter Diggle. *Time series: a biostatistical introduction*. Oxford University Press, 1990.

[6] John Fox. Aspects of the social organization and trajectory of the r project. *The R Journal*, 1(2):5, 2009.

[7] Carolo Friderico Gauss. *Theoria motvs corporvm coelestivm in sectionibvs conicis Solem ambientivm (Theory of the Motion of the Heavenly Bodies Moving about the Sun in Conic Sections)*. Hambvrgi, Svmtibvs F. Perthes et I. H. Besser, 1809. In Latin.

[8] Kurt Hornik. The comprehensive r archive network. *Wiley interdisciplinary reviews: Computational statistics*, 4(4):394–398, 2012.

[9] Ross Ihaka. R: Past and future history. *Computing Science and Statistics*, 392396, 1998.

[10] Norman L. Johnson, Samuel Kotz, and Narayanaswamy Balakrishnan. *Continuous Univariate Distributions, Volume 1*. Wiley, 1994.

[11] Siméon D. Poisson. *Probabilité des jugements en matière criminelle et en matière civile, précédées des règles générales du calcul des probabilités*. Bachelier, Paris, France, 1837.

[12] Stephen M Stigler. Poisson on the poisson distribution. *Statistics & Probability Letters*, 1(1):33–35, 1982.

[13] Stephen M Stigler. *Statistics on the table: The history of statistical concepts and methods*. Harvard University Press, 2002.

[14] The Royal Mint. The history of the coin toss, n.d. Available online at `https://www.royalmint.com/stories/celebrate/the-history-of-the-coin-toss/`.