The RBioc Book

Sean Davis University of ColoradoAnschutz School of Medicine

2024-07-26

Table of contents

List of Figures

List of Tables

Preface

Who is this book for?

- People who want to learn data science
- People who want to teach data science
- People who want to learn how to teach data science
- People who want to learn how to learn data science

Why this book?

This book is a collection of resources for learning R and Bioconductor. It is meant to be largely self-directed, but for those looking to teach data science, it can also be used as a guide for structuring a course. Material is a bit variable in terms of difficulty, prerequisites, and format which is a reflection of the organic creation of the material.

Students are encouraged to work with others to learn the material. Instructors are encouraged to use the material to create a course that is tailored to the needs of their students and to spend lots of time in 1:1 and small groups to support students in their learning. See below for additional thoughts on adult learning and how it relates to this material.

Adult learners

Adult Learning Theory, also known as Andragogy, is the concept and practice of designing, developing, and delivering instructional experiences for adult learners. It is based on the belief that adults learn differently than children, and thus, require distinct approaches to engage, motivate, and retain information (Center 2016). The term was first introduced by Malcolm Knowles, an American educator who is known for his work in adult education (Knowles, Holton, and Swanson 2005).

One of the fundamental principles of Adult Learning Theory is that adults are self-directed learners. This means that we prefer to take control of our own learning process and set personal goals for themselves. We are motivated by our desire to solve problems or gain knowledge to improve our lives (see Figure 1). As a result, educational content for adults should be relevant and applicable to real-life situations. Furthermore, adult learners should be given opportunities to actively engage in the learning process by making choices, setting goals, and evaluating their progress.

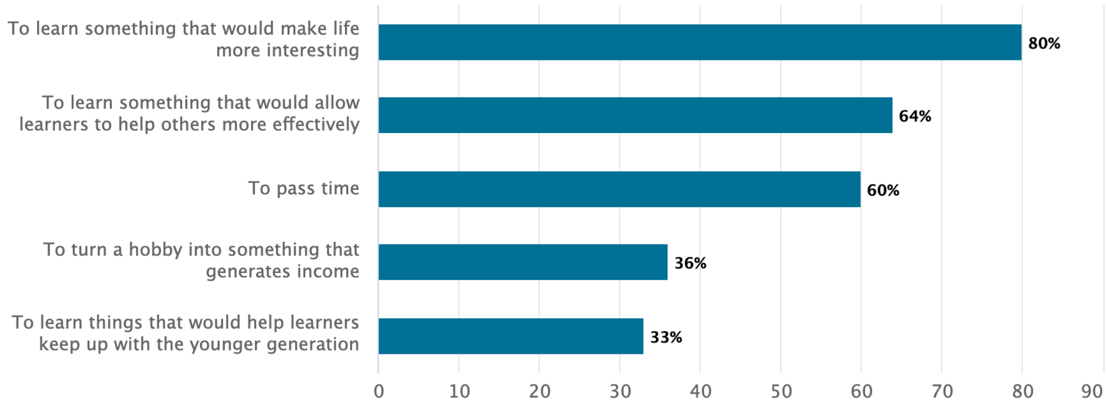


Figure 1.: Why do adults choose to learn something?

Another key aspect of Adult Learning Theory is the role of experience. We bring a wealth of experience to the learning process, which serves as a resource for new learning. We often have well-established beliefs, values, and mental models that can influence our willingness to accept new ideas and concepts. Therefore, it is essential to acknowledge and respect our shared and unique past experiences and create an environment where we all feel comfortable sharing our perspectives.

To effectively learn as a group of adult learners, it is crucial to establish a collaborative learning environment that promotes open communication and fosters trust among participants. We all appreciate and strive for a respectful and supportive atmosphere where we can express our opinions without fear of judgment. Instructors should help facilitate discussions, encourage peer-to-peer interactions, and incorporate group activities and collaboration to capitalize on the collective knowledge of participants.

Additionally, adult learners often have multiple responsibilities outside of the learning environment, such as work and family commitments. As a result, we require flexible learning opportunities that accommodate busy schedules. Offering a variety of instructional formats, such as online modules, self-paced learning, or evening classes, can help ensure that adult learners have access to education despite any time constraints.

Adult learners benefit from a learner-centered approach that focuses on the individual needs, preferences, and interests of each participant can greatly enhance the overall learning experience. In addition, we tend to be more intrinsically motivated to learn when we have a sense of autonomy and can practice and experiment (see Figure 2) with new concepts in a safe environment.

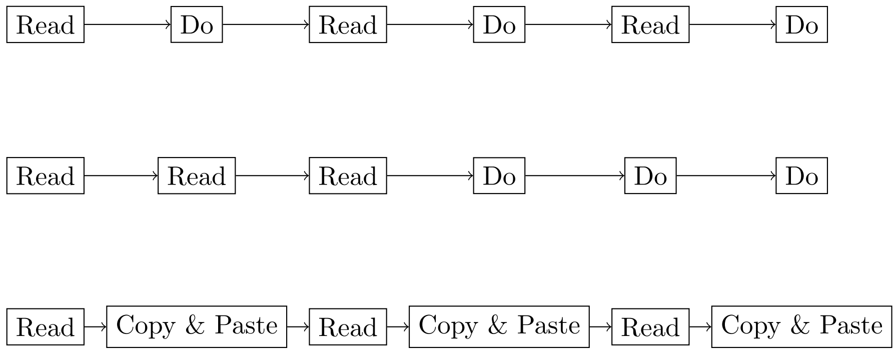


Figure 2.: How to stay stuck in data science (or anything). The “Read-Do” loop tends to deliver the best results. Too much reading between doing can be somewhat effective. Reading and simply copy-paste is probably the least effective. When working through material, experiment. Try to break things. Incorporate your own experience or applications whenever possible.

Understanding Adult Learning Theory and its principles can significantly enhance the effectiveness of teaching and learning as adults. By respecting our autonomy, acknowledging our experiences, creating a supportive learning environment, offering flexible learning opportunities, and utilizing diverse teaching methods, we can better cater to the unique needs and preferences of adult learners.

In practice, that means that we will will not be prescriptive in our approach to teaching data science. We will not tell you what to do, but rather we will provide you with a variety of options and you can choose what works best for you. We will also provide you with a variety of resources and you can choose where to focus your time. Given that we cannot possibly cover everything, we will provide you with a framework for learning and you can fill in the gaps as you see fit. A key component of our success as adult learners is to gain the confidence to ask questions and problem-solve on our own.

Part I.

Introduction

1. Introducing R and RStudio

Questions

- What is R?
- Why use R?
- Why not use R?
- Why use RStudio and how does it differ from R?

Learning Objectives

- Know advantages of analyzing data in R
- Know advantages of using RStudio
- Be able to start RStudio on your computer
- Identify the panels of the RStudio interface
- Be able to customize the RStudio layout

1.1. Introduction

In this chapter, we will discuss the basics of R and RStudio, two essential tools in genomics data analysis. We will cover the advantages of using R and RStudio, how to set up RStudio, and the different panels of the RStudio interface.

1.2. What is R?

R(https://en.wikipedia.org/wiki/R_(programming_language) is a programming language and software environment designed for statistical computing and graphics. It is widely used by statisticians, data scientists, and researchers for data analysis and visualization. R is an open-source language, which means it is free to use, modify, and distribute. Over the years, R has become particularly popular in the fields of genomics and bioinformatics, owing to its extensive libraries and powerful data manipulation capabilities.

The R language is a dialect of the S language, which was developed in the 1970s at Bell Laboratories. The first version of R was written by Robert Gentleman and Ross Ihaka and released in 1995 (see this slide deck for Ross Ihaka's take on R's history). Since then, R has been continuously developed by the R Core Team, a group of statisticians and computer scientists. The R Core Team releases a new version of R every year.

Figure 1.1.: Google trends showing the popularity of R over time based on Google searches

1.3. Why use R?

There are several reasons why R is a popular choice for data analysis, particularly in genomics and bioinformatics. These include:

1. **Open-source**: R is free to use and has a large community of developers who contribute to its growth and development. What is "open-source"?
2. **Extensive libraries**: There are thousands of R packages available for a wide range of tasks, including specialized packages for genomics and bioinformatics. These libraries have been extensively tested and ara available for free.
3. **Data manipulation**: R has powerful data manipulation capabilities, making it easy (or at least possible) to clean, process, and analyze large datasets.
4. **Graphics and visualization**: R has excellent tools for creating high-quality graphics and visualizations that can be customized to meet the specific needs of your analysis. In most cases, graphics produced by R are publication-quality.
5. **Reproducible research**: R enables you to create reproducible research by recording your analysis in a script, which can be easily shared and executed by others. In addition, R does not have a meaningful graphical user interface (GUI), which renders analysis in R much more reproducible than tools that rely on GUI interactions.
6. **Cross-platform**: R runs on Windows, Mac, and Linux (as well as more obscure systems).
7. **Interoperability with other languages**: R can interfact with FORTRAN, C, and many other languages.

8. **Scalability**: R is useful for small and large projects.

I can develop code for analysis on my Mac laptop. I can then install the *same* code on our 20k core cluster and run it in parallel on 100 samples, monitor the process, and then update a database (for example) with R when complete.

1.4. Why not use R?

- R cannot do everything.
- R is not always the "best" tool for the job.
- R will *not* hold your hand. Often, it will *slap* your hand instead.
- The documentation can be opaque (but there is documentation).
- R can drive you crazy (on a good day) or age you prematurely (on a bad one).
- Finding the right package to do the job you want to do can be challenging; worse, some contributed packages are unreliable.]{}
- R does not have a meaningfully useful graphical user interface (GUI).

1.5. R License and the Open Source Ideal

R is free (yes, totally free!) and distributed under GNU license. In particular, this license allows one to:

- Download the source code
- Modify the source code to your heart's content
- Distribute the modified source code and even charge money for it, but you must distribute the modified source code under the original GNU license]{}

This license means that R will always be available, will always be open source, and can grow organically without constraint.

1.6. RStudio

RStudio is an integrated development environment (IDE) for R. It provides a graphical user interface (GUI) for R, making it easier to write and execute R code. RStudio also provides several other useful features, including a built-in console, syntax-highlighting editor, and tools for plotting, history, debugging, workspace management, and workspace viewing. RStudio is available in both free and commercial editions; the commercial edition provides some additional features, including support for multiple sessions and enhanced debugging

1.6.1. Getting started with RStudio

To get started with RStudio, you first need to install both R and RStudio on your computer. Follow these steps:

1. Download and install R from the official R website.
2. Download and install RStudio from the official RStudio website.
3. Launch RStudio. You should see the RStudio interface with four panels.

1.6.2. The RStudio Interface

RStudio's interface consists of four panels (see Figure 1.2):

- **Console** This panel displays the R console, where you can enter and execute R commands directly. The console also shows the output of your code, error messages, and other information.
- **Source** This panel is where you write and edit your R scripts. You can create new scripts, open existing ones, and run your code from this panel.
- **Environment** This panel displays your current workspace, including all variables, data objects, and functions that you have created or loaded in your R session.
- **Plots, Packages, Help, and Viewer** These panels display plots, installed packages, help files, and web content, respectively.

Do I need to use RStudio?

No. You can use R without RStudio. However, RStudio makes it easier to write and execute R code, and it provides several useful features that are not available in the basic R console. Note that the only part of RStudio that is actually interacting with R directly is the console. The other panels are simply providing a GUI that enhances the user experience.

Customizing the RStudio Interface

You can customize the layout of RStudio to suit your preferences. To do so, go to **Tools > Global Options > Appearance**. Here, you can change the theme, font size, and panel layout. You can also resize the panels as needed to gain screen real estate (see Figure 1.3).

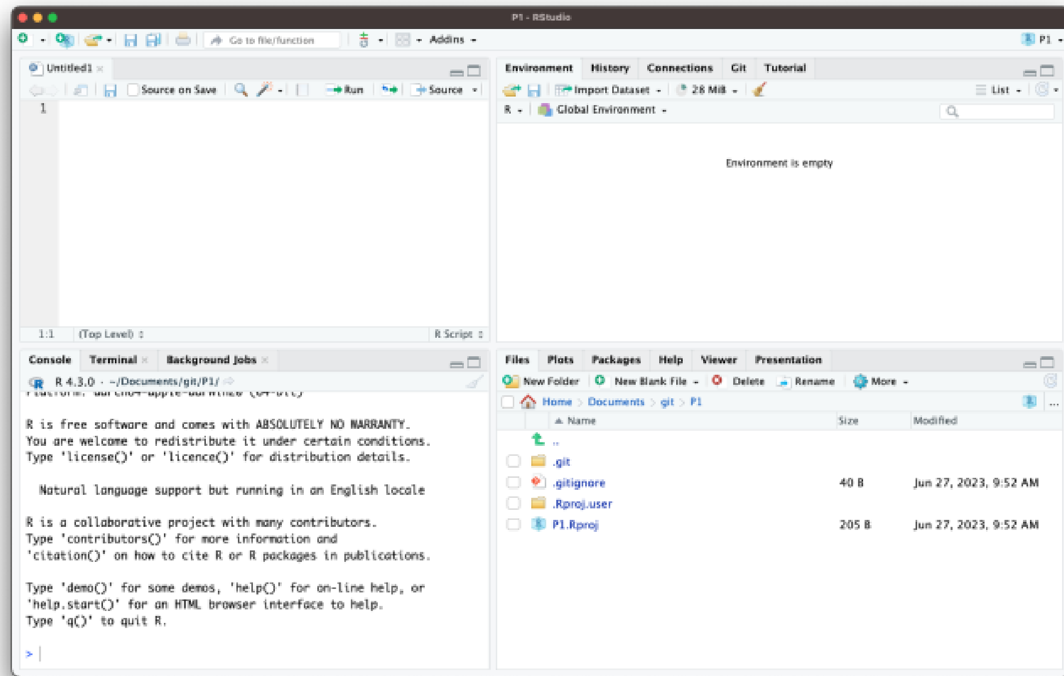


Figure 1.2.: The RStudio interface. In this layout, the **source** pane is in the upper left, the **console** is in the lower left, the **environment** panel is in the top right and the **viewer/help/files** panel is in the bottom right.

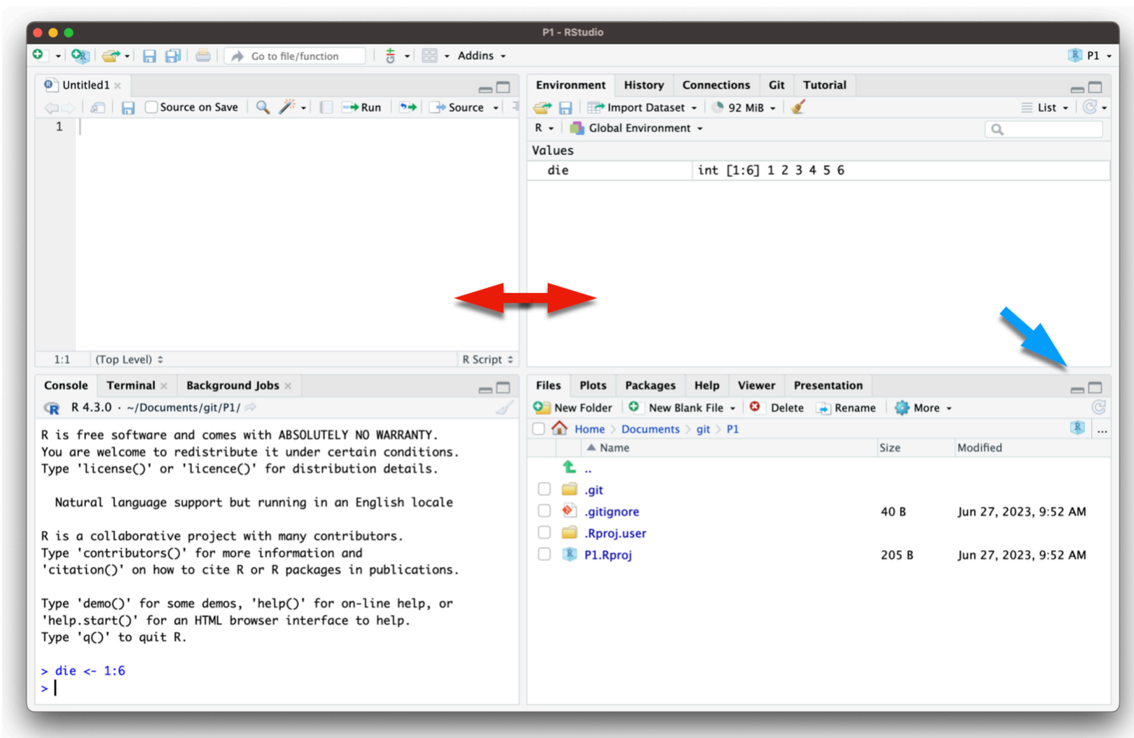


Figure 1.3.: Dealing with limited screen real estate can be a challenge, particularly when you want to open another window to, for example, view a web page. You can resize the panes by sliding the center divider (red arrows) or by clicking on the minimize/maximize buttons (see blue arrow).

In summary, R and RStudio are powerful tools for genomics data analysis. By understanding the advantages of using R and RStudio and familiarizing yourself with the RStudio interface, you can efficiently analyze and visualize your data. In the following chapters, we will delve deeper into the functionality of R, Bioconductor, and various statistical methods to help you gain a comprehensive understanding of genomics data analysis.

2. R mechanics

2.1. Learning objectives

- Be able to start R and RStudio
- Learn to interact with the R console
- Know the difference between expressions and assignment
- Recognize valid and invalid R names
- Know how to access the R help system
- Know how to assign values to variables, find what is in R memory, and remove values from R memory

2.2. Installing R

R is available for Windows, Mac, and Linux. To install R, go to the Comprehensive R Archive Network (CRAN). Click on the download link for your operating system and follow the instructions.

2.3. Installing RStudio

RStudio is an Integrated Development Environment (IDE) for R. It is available for Windows, Mac, and Linux. To install RStudio, go to the RStudio download page. Click on the download link for your operating system and follow the instructions.

2.4. Starting R

How to start R depends a bit on the operating system (Mac, Windows, Linux) and interface. In this course, we will largely be using an Integrated Development Environment (IDE) called *RStudio*, but there is nothing to prohibit using R at the command line or in some other interface (and there are a few).

2.5. *RStudio*: A Quick Tour

The RStudio interface has multiple panes. All of these panes are simply for convenience except the "Console" panel, typically in the lower left corner (by default). The console pane contains the running R interface. If you choose to run R outside RStudio, the interaction will be *identical* to working in the console pane. This is useful to keep in mind as some environments, such as a computer cluster, encourage using R without RStudio.

- Panes
- Options
- Help
- Environment, History, and Files

2.6. Interacting with R

The only meaningful way of interacting with R is by typing into the R console. At the most basic level, anything that we type at the command line will fall into one of two categories:

1. Assignments

   ```
   x = 1
   y <- 2
   ```

2. Expressions

   ```
   1 + pi + sin(42)
   ```

   ```
   [1] 3.225071
   ```

The assignment type is obvious because either the The `<-` or `=` are used. Note that when we type expressions, R will return a result. In this case, the result of R evaluating `1 + pi + sin(42)` is 3.2250711.

The standard R prompt is a ">" sign. When present, R is waiting for the next expression or assignment. If a line is not a complete R command, R will continue the next line with a "+". For example, typing the fillowing with a "Return" after the second "+" will result in R giving back a "+" on the next line, a prompt to keep typing.

```
1 + pi +
sin(3.7)
```

```
[1] 3.611757
```

R can be used as a glorified calculator by using R expressions. Mathematical operations include:

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Exponentiation: `^`
- Modulo: `%%`

The `^` operator raises the number to its left to the power of the number to its right: for example `3^2` is `9`. The modulo returns the remainder of the division of the number to the left by the number on its right, for example 5 modulo 3 or `5 %% 3` is 2.

2.6.1. Expressions

```
5 + 2
28 %% 3
3^2
5 + 4 * 4 + 4 ^ 4 / 10
```

Note that R follows order-of-operations and groupings based on parentheses.

```
5 + 4 / 9
(5 + 4) / 9
```

2.6.2. Assignment

While using R as a calculator is interesting, to do useful and interesting things, we need to assign *values* to *objects*. To create objects, we need to give it a name followed by the assignment operator `<-` (or, entirely equivalently, `=`) and the value we want to give it:

```
weight_kg <- 55
```

`<-` is the assignment operator. Assigns values on the right to objects on the left, it is like an arrow that points from the value to the object. Using an `=` is equivalent (in nearly all cases). Learn to use `<-` as it is good programming practice.

Objects can be given any name such as `x`, `current_temperature`, or `subject_id` (see below). You want your object names to be explicit and not too long. They cannot start with a number (`2x` is not valid but `x2` is). R is case sensitive (e.g., `weight_kg` is different from `Weight_kg`). There are some names that cannot be used because they represent the names of fundamental functions in R (e.g., `if`, `else`, `for`, see here for a complete list). In general, even if it's allowed, it's best to not use other function names, which we'll get into shortly (e.g., `c`, `T`, `mean`, `data`, `df`, `weights`). When in doubt, check the help to see if the name is already in use. It's also best to avoid dots (`.`) within a variable name as in `my.dataset`. It is also recommended to use nouns for variable names, and verbs for function names.

When assigning a value to an object, R does not print anything. You can force to print the value by typing the name:

```
weight_kg
```

```
[1] 55
```

Now that R has `weight_kg` in memory, which R refers to as the "global environment", we can do arithmetic with it. For instance, we may want to convert this weight in pounds (weight in pounds is 2.2 times the weight in kg).

```
2.2 * weight_kg
```

```
[1] 121
```

We can also change a variable's value by assigning it a new one:

```
weight_kg <- 57.5
2.2 * weight_kg
```

```
[1] 126.5
```

This means that assigning a value to one variable does not change the values of other variables. For example, let's store the animal's weight in pounds in a variable.

```
weight_lb <- 2.2 * weight_kg
```

and then change `weight_kg` to 100.

```
weight_kg <- 100
```

What do you think is the current content of the object `weight_lb`, 126.5 or 220?

You can see what objects (variables) are stored by viewing the Environment tab in Rstudio. You can also use the `ls()` function. You can remove objects (variables) with the `rm()` function. You can do this one at a time or remove several objects at once. You can also use the little broom button in your environment pane to remove everything from your environment.

```
ls()
rm(weight_lb, weight_kg)
ls()
```

What happens when you type the following, now?

```
weight_lb # oops! you should get an error because weight_lb no longer exists!
```

2.7. Rules for Names in R

R allows users to assign names to objects such as variables, functions, and even dimensions of data. However, these names must follow a few rules.

- Names may contain any combination of letters, numbers, underscore, and "."
- Names may not start with numbers, underscore.
- R names are case-sensitive.

Examples of valid R names include:

```
pi
x
camelCaps
my_stuff
MY_Stuff
```

```
this.is.the.name.of.the.man
ABC123
abc1234asdf
.hi
```

2.8. Resources for Getting Help

There is extensive built-in help and documentation within R. A separate page contains a collection of additional resources.

If the name of the function or object on which help is sought is known, the following approaches with the name of the function or object will be helpful. For a concrete example, examine the help for the `print` method.

```
help(print)
help('print')
?print
```

If the name of the function or object on which help is sought is *not* known, the following from within R will be helpful.

```
help.search('microarray')
RSiteSearch('microarray')
apropos('histogram')
```

There are also tons of online resources that Google will include in searches if online searching feels more appropriate.

I strongly recommend using `help("newfunction"")` for all functions that are new or unfamiliar to you.

There are also many open and free resources and reference guides for R.

- Quick-R: a quick online reference for data input, basic statistics and plots
- R reference card PDF by Tom Short
- Rstudio cheatsheets

3. Up and Running with R

In this chapter, we're going to get an introduction to the R language, so we can dive right into programming. We're going to create a pair of virtual dice that can generate random numbers. No need to worry if you're new to programming. We'll return to many of the concepts here in more detail later.

To simulate a pair of dice, we need to break down each die into its essential features. A die can only show one of six numbers: 1, 2, 3, 4, 5, and 6. We can capture the die's essential characteristics by saving these numbers as a group of values in the computer. Let's save these numbers first and then figure out a way to "roll" our virtual die.

3.1. The R User Interface

The RStudio interface is simple. You type R code into the bottom line of the RStudio console pane and then click Enter to run it. The code you type is called a *command*, because it will command your computer to do something for you. The line you type it into is called the *command line*.

When you type a command at the prompt and hit Enter, your computer executes the command and shows you the results. Then RStudio displays a fresh prompt for your next command. For example, if you type `1 + 1` and hit Enter, RStudio will display:

```
> 1 + 1
[1] 2
>
```

You'll notice that a `[1]` appears next to your result. R is just letting you know that this line begins with the first value in your result. Some commands return more than one value, and their results may fill up multiple lines. For example, the command `100:130` returns 31 values; it creates a sequence of integers from 100 to 130. Notice that new bracketed numbers appear at the start of the second and third lines of output. These numbers just mean that the second line begins with the 14th value in the result, and the third line begins with the 25th value. You can mostly ignore the numbers that appear in brackets:

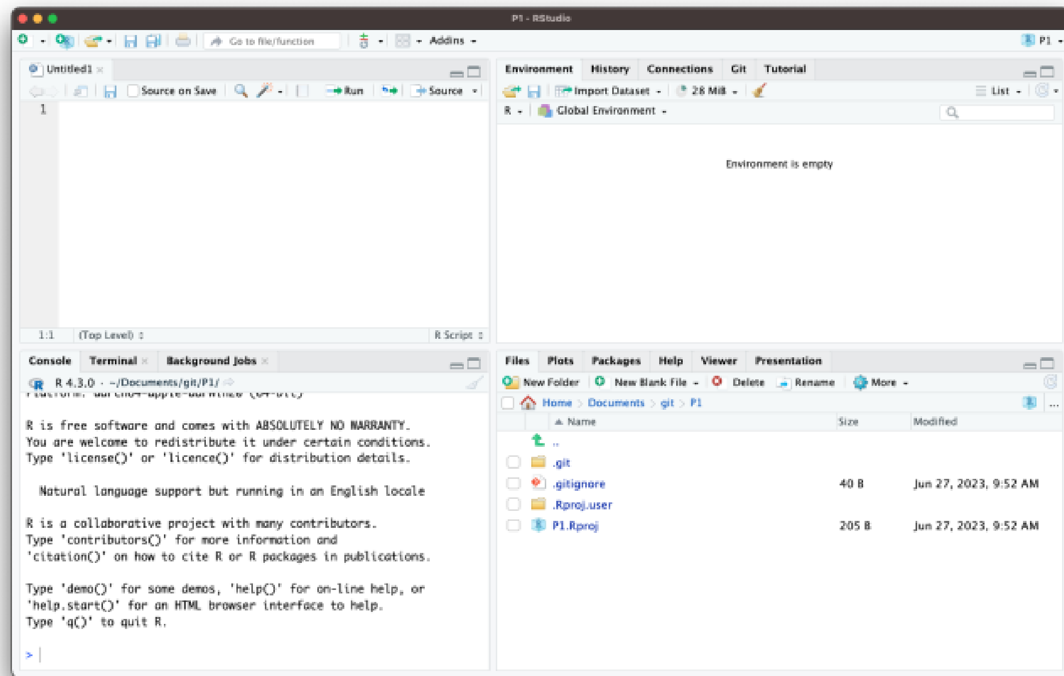


Figure 3.1.: Your computer does your bidding when you type R commands at the prompt in the bottom line of the console pane. Don't forget to hit the Enter key. When you first open RStudio, the console appears in the pane on your left, but you can change this with **File** > **Tools** > **Global Options** in the menu bar.

```
> 100:130
 [1] 100 101 102 103 104 105 106 107 108 109 110 111 112
[14] 113 114 115 116 117 118 119 120 121 122 123 124 125
[25] 126 127 128 129 130
```

> **Tip**
>
> The colon operator (`:`) returns every integer between two integers. It is an easy way to create a sequence of numbers.

> **When do we compile?**
>
> In some languages, like C, Java, and FORTRAN, you have to compile your human-readable code into machine-readable code (often 1s and 0s) before you can run it. If you've programmed in such a language before, you may wonder whether you have to compile your R code before you can use it. The answer is no. R is a dynamic programming language, which means R automatically interprets your code as you run it.

If you type an incomplete command and press Enter, R will display a `+` prompt, which means R is waiting for you to type the rest of your command. Either finish the command or hit Escape to start over:

```
> 5 -
+
+ 1
[1] 4
```

If you type a command that R doesn't recognize, R will return an error message. If you ever see an error message, don't panic. R is just telling you that your computer couldn't understand or do what you asked it to do. You can then try a different command at the next prompt:

```
> 3 % 5
Error: unexpected input in "3 % 5"
>
```

Tip

Whenever you get an error message in R, consider googling the error message. You'll often find that someone else has had the same problem and has posted a solution online. Simply cutting-and-pasting the error message into a search engine will often work

Once you get the hang of the command line, you can easily do anything in R that you would do with a calculator. For example, you could do some basic arithmetic:

```
2 * 3
```

```
[1] 6
```

```
4 - 1
```

```
[1] 3
```

```
# this obeys order-of-operations
6 / (4 - 1)
```

```
[1] 2
```

Tip

R treats the hashtag character, `#`, in a special way; R will not run anything that follows a hashtag on a line. This makes hashtags very useful for adding comments and annotations to your code. Humans will be able to read the comments, but your computer will pass over them. The hashtag is known as the *commenting symbol* in R.

! Cancelling commands

Some R commands may take a long time to run. You can cancel a command once it has begun by pressing ctrl + c or by clicking the "stop sign" if it is available in Rstudio. Note that it may also take R a long time to cancel the command.

3.1.1. An exercise

That's the basic interface for executing R code in RStudio. Think you have it? If so, try doing these simple tasks. If you execute everything correctly, you should end up with the same number that you started with:

1. Choose any number and add 2 to it.
2. Multiply the result by 3.
3. Subtract 6 from the answer.
4. Divide what you get by 3.

```
10 + 2
```

```
[1] 12
```

```
12 * 3
```

```
[1] 36
```

```
36 - 6
```

```
[1] 30
```

```
30 / 3
```

```
[1] 10
```

3.2. Objects

Now that you know how to use R, let's use it to make a virtual die. The `:` operator from a couple of pages ago gives you a nice way to create a group of numbers from one to six. The `:` operator returns its results as a **vector** (we are going to work with vectors in more detail), a one-dimensional set of numbers:

```
1:6
## 1 2 3 4 5 6
```

That's all there is to how a virtual die looks! But you are not done yet. Running `1:6` generated a vector of numbers for you to see, but it didn't save that vector anywhere for later use. If we want to use those numbers again, we'll have to ask your computer to save them somewhere. You can do that by creating an R *object*.

R lets you save data by storing it inside an R object. What is an object? Just a name that you can use to call up stored data. For example, you can save data into an object like *a* or *b*. Wherever R encounters the object, it will replace it with the data saved inside, like so:

```
a <- 1
a
```

```
[1] 1
```

```
a + 2
```

```
[1] 3
```

What just happened?

1. To create an R object, choose a name and then use the less-than symbol, `<`, followed by a minus sign, `-`, to save data into it. This combination looks like an arrow, `<-`. R will make an object, give it your name, and store in it whatever follows the arrow. So `a <- 1` stores `1` in an object named `a`.
2. When you ask R what's in `a`, R tells you on the next line.
3. You can use your object in new R commands, too. Since `a` previously stored the value of `1`, you're now adding `1` to `2`.

Assignment vs expressions

Everything that you type into the R console can be assigned to one of two categories:

- Assignments
- Expressions

An expression is a command that tells R to do something. For example, `1 + 2` is an expression that tells R to add `1` and `2`. When you type an expression into the R console, R will evaluate the expression and return the result. For example, if you type `1 + 2` into the R console, R will return `3`. Expressions can have "side effects" but they don't explicitly result in anything being added to R memory.

```
5 + 2
```

```
[1] 7
```

```
28 %% 3
```

```
[1] 1
```

```
3^2
```

```
[1] 9
```

```
5 + 4 * 4 + 4 ^ 4 / 10
```

```
[1] 46.6
```

While using R as a calculator is interesting, to do useful and interesting things, we need to assign values to objects. To create objects, we need to give it a name followed by the assignment operator `<-` (or, entirely equivalently, `=`) and the value we want to give it:

```
weight_kg <- 55
```

So, for another example, the following code would create an object named `die` that contains the numbers one through six. To see what is stored in an object, just type the object's name by itself:

```
die <- 1:6
die
```

```
[1] 1 2 3 4 5 6
```

When you create an object, the object will appear in the environment pane of RStudio, as shown in Figure 3.2. This pane will show you all of the objects you've created since opening RStudio.

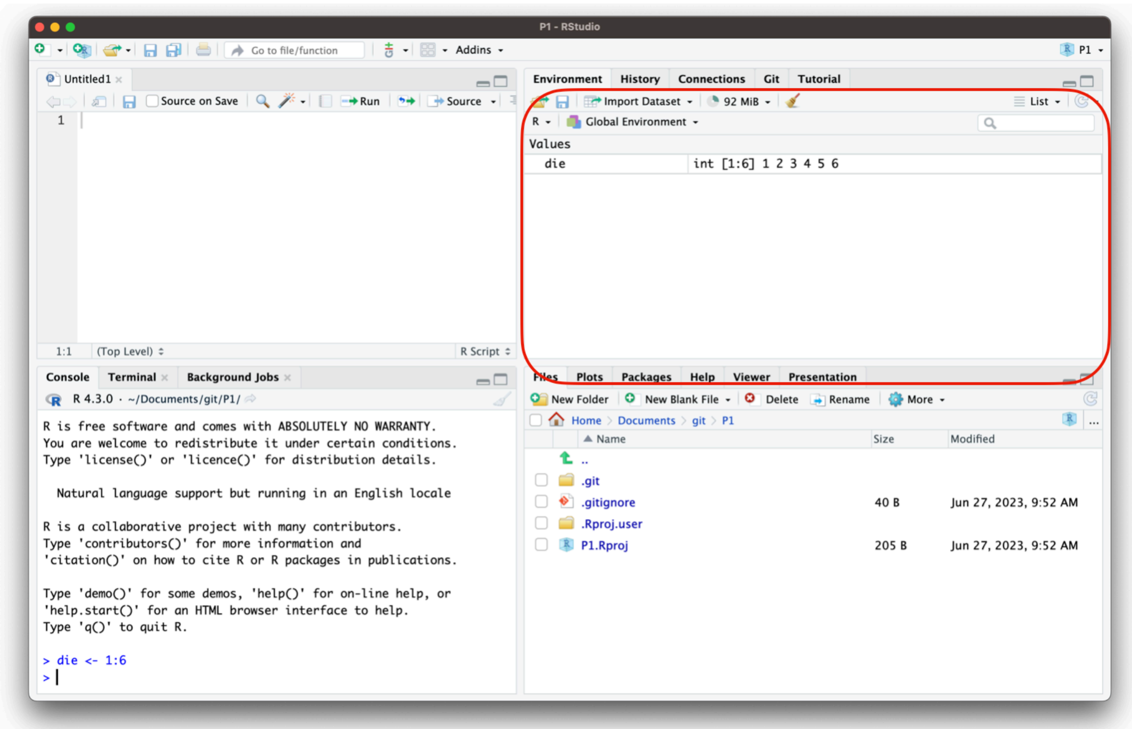


Figure 3.2.: Assignment creates an object in the environment pane.

You can name an object in R almost anything you want, but there are a few rules. First, a name cannot start with a number. Second, a name cannot use some special symbols, like `^`, `!`, `$`, `@`, `+`, `-`, `/`, or `*`:

Good names	Names that cause errors
a	1trial
b	$
FOO	^mean
my_var	2nd
.day	!bad

Capitalization matters

R is case-sensitive, so `name` and `Name` will refer to different objects:

```
> Name = 0
> Name + 1
[1] 1
> name + 1
Error: object 'name' not found
```

The error above is a common one!

Finally, R will overwrite any previous information stored in an object without asking you for permission. So, it is a good idea to *not* use names that are already taken:

```
my_number <- 1
my_number
```

```
[1] 1
```

```
my_number <- 999
my_number
```

```
[1] 999
```

You can see which object names you have already used with the function `ls`:

```
ls()
```

Your environment will contain different names than mine, because you have probably created different objects.

You can also see which names you have used by examining RStudio's environment pane.

We now have a virtual die that is stored in the computer's memory and which has a name that we can use to refer to it. You can access it whenever you like by typing the word *die*.

So what can you do with this die? Quite a lot. R will replace an object with its contents whenever the object's name appears in a command. So, for example, you can do all sorts of math with the die. Math isn't so helpful for rolling dice, but manipulating sets of numbers will be your stock and trade as a data scientist. So let's take a look at how to do that:

```
die - 1
```

```
[1] 0 1 2 3 4 5
```

```
die / 2
```

```
[1] 0.5 1.0 1.5 2.0 2.5 3.0
```

```
die * die
```

```
[1]  1  4  9 16 25 36
```

R uses *element-wise execution* when working with a *vector* like `die`. When you manipulate a set of numbers, R will apply the same operation to each element in the set. So for example, when you run *die - 1*, R subtracts one from each element of `die`.

When you use two or more vectors in an operation, R will line up the vectors and perform a sequence of individual operations. For example, when you run *die * die*, R lines up the two `die` vectors and then multiplies the first element of vector 1 by the first element of vector 2. R then multiplies the second element of vector 1 by the second element of vector 2, and so on, until every element has been multiplied. The result will be a new vector the same length as the first two {Figure 3.3}.

If you give R two vectors of unequal lengths, R will repeat the shorter vector until it is as long as the longer vector, and then do the math, as shown in Figure 3.4. This isn't a

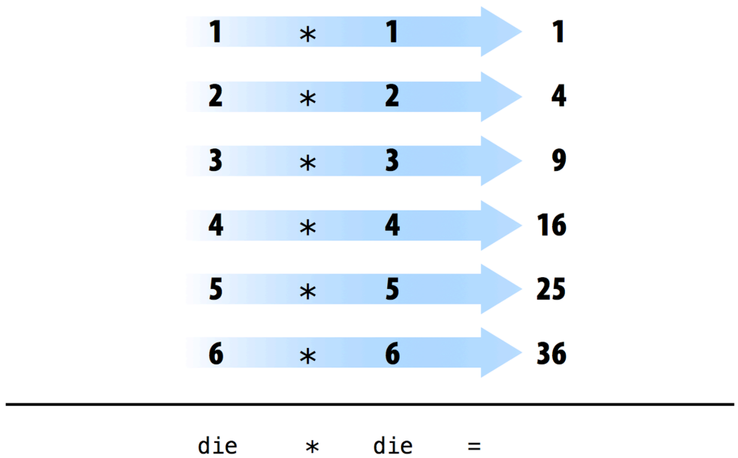


Figure 3.3.: "When R performs element-wise execution, it matches up vectors and then manipulates each pair of elements independently."

permanent change–the shorter vector will be its original size after R does the math. If the length of the short vector does not divide evenly into the length of the long vector, R will return a warning message. This behavior is known as *vector recycling*, and it helps R do element-wise operations:

```
1:2
```

```
[1] 1 2
```

```
1:4
```

```
[1] 1 2 3 4
```

```
die
```

```
[1] 1 2 3 4 5 6
```

```
die + 1:2
```

```
[1] 2 4 4 6 6 8
```

```
die + 1:4
```

```
Warning in die + 1:4: longer object length is not a multiple of shorter object
length
```

```
[1] 2 4 6 8 6 8
```

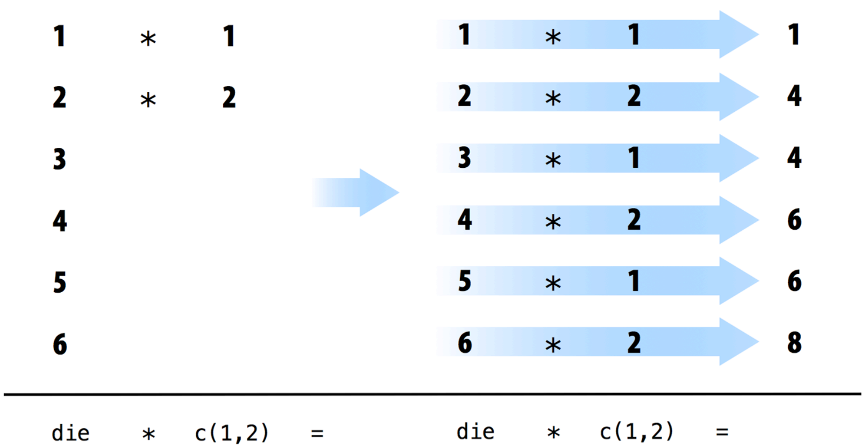


Figure 3.4.: "R will repeat a short vector to do element-wise operations with two vectors of uneven lengths."

Element-wise operations are a very useful feature in R because they manipulate groups of values in an orderly way. When you start working with data sets, element-wise operations will ensure that values from one observation or case are only paired with values from the same observation or case. Element-wise operations also make it easier to write your own programs and functions in R.

! Element-wise operations are not matrix operations

It is important to know that operations with vectors are not the same that you might expect if you are expecting R to perform "matrix" operations. R can do inner multiplication with the `%*%` operator and outer multiplication with the `%o%` operator:

```
# Inner product (1*1 + 2*2 + 3*3 + 4*4 + 5*5 + 6*6)
die %*% die
# Outer product
die %o% die
```

Now that you can do math with your `die` object, let's look at how you could "roll" it. Rolling your die will require something more sophisticated than basic arithmetic; you'll need to randomly select one of the die's values. And for that, you will need a *function*.

3.3. Functions

R has many functions and puts them all at our disposal. We can use functions to do simple and sophisticated tasks. For example, we can round a number with the `round` function, or calculate its factorial with the `factorial` function. Using a function is pretty simple. Just write the name of the function and then the data you want the function to operate on in parentheses:

```
round(3.1415)
```

```
[1] 3
```

```
factorial(3)
```

```
[1] 6
```

The data that you pass into the function is called the function's *argument*. The argument can be raw data, an R object, or even the results of another R function. In this last case, R will work from the innermost function to the outermost Figure 3.5.

```
mean(1:6)
```

```
[1] 3.5
```

```
mean(die)
```

```
[1] 3.5
```

```
round(mean(die))
```

```
[1] 4
```

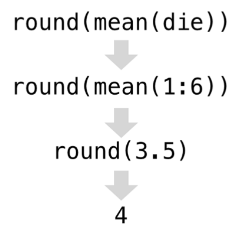


Figure 3.5.: "When you link functions together, R will resolve them from the innermost operation to the outermost. Here R first looks up die, then calculates the mean of one through six, then rounds the mean."

Returning to our die, we can use the `sample` function to randomly select one of the die's values; in other words, the `sample` function can simulate rolling the `die`.

The `sample` function takes *two* arguments: a vector named `x` and a number named `size`. `sample` will return `size` elements from the vector:

```
sample(x = 1:4, size = 2)
```

```
[1] 3 4
```

To roll your die and get a number back, set `x` to `die` and sample one element from it. You'll get a new (maybe different) number each time you roll it:

```
sample(x = die, size = 1)
```

```
[1] 5
```

```
sample(x = die, size = 1)
```

```
[1] 5
```

```
sample(x = die, size = 1)
```

```
[1] 3
```

Many R functions take multiple arguments that help them do their job. You can give a function as many arguments as you like as long as you separate each argument with a comma.

You may have noticed that I set `die` and `1` equal to the names of the arguments in `sample`, `x` and `size`. Every argument in every R function has a name. You can specify which data should be assigned to which argument by setting a name equal to data, as in the preceding code. This becomes important as you begin to pass multiple arguments to the same function; names help you avoid passing the wrong data to the wrong argument. However, using names is optional. You will notice that R users do not often use the name of the first argument in a function. So you might see the previous code written as:

```
sample(die, size = 1)
```

```
[1] 4
```

Often, the name of the first argument is not very descriptive, and it is usually obvious what the first piece of data refers to anyways.

But how do you know which argument names to use? If you try to use a name that a function does not expect, you will likely get an error:

```
round(3.1415, corners = 2)
## Error in round(3.1415, corners = 2) : unused argument(s) (corners = 2)
```

If you're not sure which names to use with a function, you can look up the function's arguments with `args`. To do this, place the name of the function in the parentheses behind `args`. For example, you can see that the `round` function takes two arguments, one named `x` and one named `digits`:

```
args(round)
```

```
function (x, digits = 0, ...)
NULL
```

Did you notice that `args` shows that the `digits` argument of `round` is already set to 0? Frequently, an R function will take optional arguments like `digits`. These arguments are considered optional because they come with a default value. You can pass a new value to an optional argument if you want, and R will use the default value if you do not. For example, `round` will round your number to 0 digits past the decimal point by default. To override the default, supply your own value for `digits`:

```
round(3.1415)
```

```
[1] 3
```

```
round(3.1415, digits = 2)
```

```
[1] 3.14
```

```
# pi happens to be a built-in value in R
pi
```

```
[1] 3.141593
```

```
round(pi)
```

```
[1] 3
```

You should write out the names of each argument after the first one or two when you call a function with multiple arguments. Why? First, this will help you and others understand your code. It is usually obvious which argument your first input refers to (and sometimes the second input as well). However, you'd need a large memory to remember the third and fourth arguments of every R function. Second, and more importantly, writing out argument names prevents errors.

If you do not write out the names of your arguments, R will match your values to the arguments in your function by order. For example, in the following code, the first value, `die`, will be matched to the first argument of `sample`, which is named `x`. The next value, `1`, will be matched to the next argument, `size`:

```
sample(die, 1)
```

```
[1] 3
```

As you provide more arguments, it becomes more likely that your order and R's order may not align. As a result, values may get passed to the wrong argument. Argument names prevent this. R will always match a value to its argument name, no matter where it appears in the order of arguments:

```
sample(size = 1, x = die)
```

```
[1] 3
```

3.3.1. Sample with Replacement

If you set `size = 2`, you can *almost* simulate a pair of dice. Before we run that code, think for a minute why that might be the case. `sample` will return two numbers, one for each die:

```
sample(die, size = 2)
```

```
[1] 1 5
```

I said this "almost" works because this method does something funny. If you use it many times, you'll notice that the second die never has the same value as the first die, which means you'll never roll something like a pair of threes or snake eyes. What is going on?

By default, `sample` builds a sample *without replacement*. To see what this means, imagine that `sample` places all of the values of `die` in a jar or urn. Then imagine that `sample` reaches into the jar and pulls out values one by one to build its sample. Once a value has been drawn from the jar, `sample` sets it aside. The value doesn't go back into the jar, so it cannot be drawn again. So if `sample` selects a six on its first draw, it will not be able to select a six on the second draw; six is no longer in the jar to be selected. Although `sample` creates its sample electronically, it follows this seemingly physical behavior.

One side effect of this behavior is that each draw depends on the draws that come before it. In the real world, however, when you roll a pair of dice, each die is independent of the other. If the first die comes up six, it does not prevent the second die from coming up six. In fact, it doesn't influence the second die in any way whatsoever. You can recreate this behavior in `sample` by adding the argument `replace = TRUE`:

```
sample(die, size = 2, replace = TRUE)
```

```
[1] 3 4
```

The argument `replace = TRUE` causes `sample` to sample *with replacement*. Our jar example provides a good way to understand the difference between sampling with replacement and without. When `sample` uses replacement, it draws a value from the jar and records the value. Then it puts the value back into the jar. In other words, `sample` *replaces* each value after each draw. As a result, `sample` may select the same value on the second draw. Each value has a chance of being selected each time. It is as if every draw were the first draw.

Sampling with replacement is an easy way to create *independent random samples*. Each value in your sample will be a sample of size one that is independent of the other values. This is the correct way to simulate a pair of dice:

```
sample(die, size = 2, replace = TRUE)
```

```
[1] 4 2
```

Congratulate yourself; you've just run your first simulation in R! You now have a method for simulating the result of rolling a pair of dice. If you want to add up the dice, you can feed your result straight into the `sum` function:

```
dice <- sample(die, size = 2, replace = TRUE)
dice
```

```
[1] 4 5
```

```
sum(dice)
```

```
[1] 9
```

What would happen if you call `dice` multiple times? Would R generate a new pair of dice values each time? Let's give it a try:

```
dice
```

```
[1] 4 5
```

```
dice
```

```
[1] 4 5
```

```
dice
```

```
[1] 4 5
```

The name `dice` refers to a *vector* of two numbers. Calling more than once does not change the favlue. Each time you call `dice`, R will show you the result of that one time you called `sample` and saved the output to `dice`. R won't rerun `sample(die, 2, replace = TRUE)` to create a new roll of the dice. Once you save a set of results to an R object, those results do not change.

However, it *would* be convenient to have an object that can re-roll the dice whenever you call it. You can make such an object by writing your own R function.

3.4. Writing Your Own Functions

To recap, you already have working R code that simulates rolling a pair of dice:

```
die <- 1:6
dice <- sample(die, size = 2, replace = TRUE)
sum(dice)
```

```
[1] 9
```

You can retype this code into the console anytime you want to re-roll your dice. However, this is an awkward way to work with the code. It would be easier to use your code if you wrapped it into its own function, which is exactly what we'll do now. We're going to write a function named `roll` that you can use to roll your virtual dice. When you're finished, the function will work like this: each time you call `roll()`, R will return the sum of rolling two dice:

```
roll()
## 8

roll()
## 3

roll()
## 7
```

Functions may seem mysterious or fancy, but they are *just another type of R object*. Instead of containing data, they contain code. This code is stored in a special format that makes it easy to reuse the code in new situations. You can write your own functions by recreating this format.

3.4.1. The Function Constructor

Every function in R has three basic parts: a name, a body of code, and a set of arguments. To make your own function, you need to replicate these parts and store them in an R object, which you can do with the `function` function. To do this, call `function()` and follow it with a pair of braces, `{}`:

```
my_function <- function() {}
```

This function, as written, doesn't do anything (yet). However, it is a valid function. You can call it by typing its name followed by an open and closed parenthesis:

```
my_function()
```

```
NULL
```

`function` will build a function out of whatever R code you place between the braces. For example, you can turn your dice code into a function by calling:

```
roll <- function() {
  die <- 1:6
  dice <- sample(die, size = 2, replace = TRUE)
  sum(dice)
}
```

Indentation and readability

Notice each line of code between the braces is indented. This makes the code easier to read but has no impact on how the code runs. R ignores spaces and line breaks and executes one complete expression at a time. Note that in other languages like python, spacing is extremely important and part of the language.

Just hit the Enter key between each line after the first brace, `{`. R will wait for you to type the last brace, `}`, before it responds.

Don't forget to save the output of `function` to an R object. This object will become your new function. To use it, write the object's name followed by an open and closed parenthesis:

```
roll()
```

```
[1] 6
```

You can think of the parentheses as the "trigger" that causes R to run the function. If you type in a function's name *without* the parentheses, R will show you the code that is stored inside the function. If you type in the name *with* the parentheses, R will run that code:

```
roll
```

```
function() {
  die <- 1:6
  dice <- sample(die, size = 2, replace = TRUE)
  sum(dice)
}
```

```
roll()
```

```
[1] 6
```

The code that you place inside your function is known as the *body* of the function. When you run a function in R, R will execute all of the code in the body and then return the result of the last line of code. If the last line of code doesn't return a value, neither will your function, so you want to ensure that your final line of code returns a value. One way to check this is to think about what would happen if you ran the body of code line by line in the command line. Would R display a result after the last line, or would it not?

Here's some code that would display a result:

```
dice
1 + 1
sqrt(2)
```

And here's some code that would not:

```
dice <- sample(die, size = 2, replace = TRUE)
two <- 1 + 1
a <- sqrt(2)
```

Again, this is just showing the distinction between expressions and assignments.

3.5. Arguments

What if we removed one line of code from our function and changed the name `die` to `bones` (just a name–don't think of it as important), like this?

```
roll2 <- function() {
  dice <- sample(bones, size = 2, replace = TRUE)
  sum(dice)
}
```

Now I'll get an error when I run the function. The function **needs** the object `bones` to do its job, but there is no object named `bones` to be found (you can check by typing `ls()` which will show you the names in the environment, or memory).

```
roll2()
## Error in sample(bones, size = 2, replace = TRUE) :
##   object 'bones' not found
```

You can supply `bones` when you call `roll2` if you make `bones` an argument of the function. To do this, put the name `bones` in the parentheses that follow `function` when you define `roll2`:

```
roll2 <- function(bones) {
  dice <- sample(bones, size = 2, replace = TRUE)
  sum(dice)
}
```

Now `roll2` will work as long as you supply `bones` when you call the function. You can take advantage of this to roll different types of dice each time you call `roll2`.

Remember, we're rolling pairs of dice:

```
roll2(bones = 1:4)
```

```
[1] 4
```

```
roll2(bones = 1:6)
```

```
[1] 5
```

```
roll2(1:20)
```

```
[1] 23
```

Notice that `roll2` will still give an error if you do not supply a value for the `bones` argument when you call `roll2`:

```
roll2()
## Error in sample(bones, size = 2, replace = TRUE) :
##   argument "bones" is missing, with no default
```

You can prevent this error by giving the `bones` argument a default value. To do this, set `bones` equal to a value when you define `roll2`:

```
roll2 <- function(bones = 1:6) {
  dice <- sample(bones, size = 2, replace = TRUE)
  sum(dice)
}
```

Now you can supply a new value for `bones` if you like, and `roll2` will use the default if you do not:

```
roll2()
```

```
[1] 6
```

You can give your functions as many arguments as you like. Just list their names, separated by commas, in the parentheses that follow `function`. When the function is run, R will replace each argument name in the function body with the value that the user supplies for the argument. If the user does not supply a value, R will replace the argument name with the argument's default value (if you defined one).

To summarize, `function` helps you construct your own R functions. You create a body of code for your function to run by writing code between the braces that follow `function`. You create arguments for your function to use by supplying their names in the parentheses that follow `function`. Finally, you give your function a name by saving its output to an R object, as shown in Figure 3.6.

Once you've created your function, R will treat it like every other function in R. Think about how useful this is. Have you ever tried to create a new Excel option and add it to Microsoft's menu bar? Or a new slide animation and add it to Powerpoint's options? When you work with a programming language, you can do these types of things. As you learn to program in R, you will be able to create new, customized, reproducible tools for yourself whenever you like.

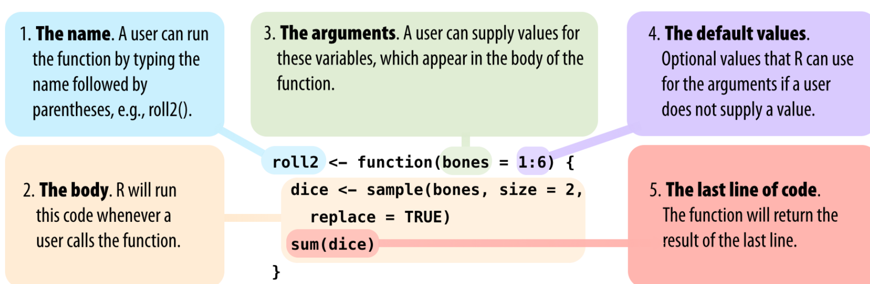


Figure 3.6.: "Every function in R has the same parts, and you can use `function` to create these parts. Assign the result to a name, so you can call the function later."

3.6. Scripts

Scripts are code that are saved for later reuse or editing. An R script is just a plain text file that you save R code in. You can open an R script in RStudio by going to **File > New File > R script** in the menu bar. RStudio will then open a fresh script above your console pane, as shown in Figure 3.7.

I strongly encourage you to write and edit all of your R code in a script before you run it in the console. Why? This habit creates a reproducible record of your work. When you're finished for the day, you can save your script and then use it to rerun your entire analysis the next day. Scripts are also very handy for editing and proofreading your code, and they make a nice copy of your work to share with others. To save a script, click the scripts pane, and then go to **File > Save As** in the menu bar.

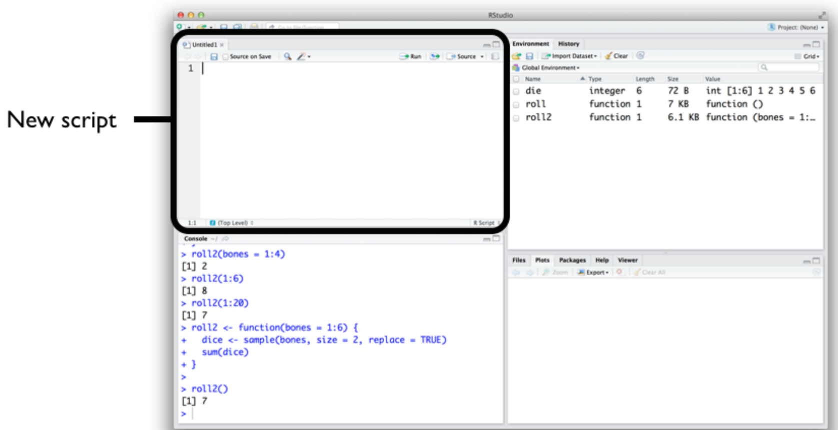


Figure 3.7.: "When you open an R Script (File > New File > R Script in the menu bar), RStudio creates a fourth pane (or puts a new tab in the existing pane) above the console where you can write and edit your code."

RStudio comes with many built-in features that make it easy to work with scripts. First, you can automatically execute a line of code in a script by clicking the Run button at the top of the editor panel.

R will run whichever line of code your cursor is on. If you have a whole section highlighted, R will run the highlighted code. Alternatively, you can run the entire script by clicking the Source button. Don't like clicking buttons? You can use Control + Return as a shortcut for the Run button. On Macs, that would be Command + Return.

If you're not convinced about scripts, you soon will be. It becomes a pain to write multi-line code in the console's single-line command line. Let's avoid that headache and open your first script now before we move to the next chapter.

> Tip
>
> **Extract function**
> RStudio comes with a tool that can help you build functions. To use it, highlight the lines of code in your R script that you want to turn into a function. Then click `Code > Extract Function` in the menu bar. RStudio will ask you for a function name to use and then wrap your code in a `function` call. It will scan the code for undefined variables and use these as arguments.
> You may want to double-check RStudio's work. It assumes that your code is correct, so if it does something surprising, you may have a problem in your code.

3.7. Summary

We've covered a lot of ground already. You now have a virtual die stored in your computer's memory, as well as your own R function that rolls a pair of dice. You've also begun speaking the R language.

The two most important components of the R language are objects, which store data, and functions, which manipulate data. R also uses a host of operators like `+`, `-`, `*`, `/`, and `<-` to do basic tasks. As a data scientist, you will use R objects to store data in your computer's memory, and you will use functions to automate tasks and do complicated calculations.

4. Packages and more dice

We now have code that allows us to roll two dice and add the results together. To keep things interesting, let's aim to weight the dice so that we can fool our friends into thinking we are lucky.

First, though, we should prove to ourselves that our dice are fair. We can investigate the behavior of our dice using two powerful and general tools;

- Simulation (or repetition or repeated sampling)
- Visualization

For the repetition part of things, we will use a built-in R function, `replicate`. For visualization, we are going to use a convenient plotting function, `qplot`. However, `qplot` does not come built into R. We must install a *package* to gain access to it.

4.1. Packages

R is a powerful language for data science and programming, allowing beginners and experts alike to manipulate, analyze, and visualize data effectively. One of the most appealing features of R is its extensive library of packages, which are essential tools for expanding its capabilities and streamlining the coding process.

An R package is a collection of reusable functions, datasets, and compiled code created by other users and developers to extend the functionality of the base R language. These packages cover a wide range of applications, such as data manipulation, statistical analysis, machine learning, and data visualization. By utilizing existing R packages, you can leverage the expertise of others and save time by avoiding the need to create custom functions from scratch.

Using others' R packages is incredibly beneficial as it allows you to take advantage of the collective knowledge of the R community. Developers often create packages to address specific challenges, optimize performance, or implement popular algorithms or methodologies. By incorporating these packages into your projects, you can enhance your productivity, reduce development time, and ensure that you are using well-tested and reliable code.

4.1.1. install.packages

To install an R package, you can use the `install.packages()` function in the R console or script. For example, to install the popular data manipulation package "dplyr," simply type install.packages("dplyr"). This command will download the package from the Comprehensive R Archive Network (CRAN) and install it on your local machine. Keep in mind that you only need to install a package once, unless you want to update it to a newer version.

In our case, we want to install the **ggplot2** package.

```
install.packages('ggplot2')
```

4.1.2. library

After installing an R package, you will need to load it into your R session before using its functions. To load a package, use the `library()` function followed by the package name, such as `library(dplyr)`. Loading a package makes its functions and datasets available for use in your current R session. Note that you need to load a package every time you start a new R session.

```
library(ggplot2)
```

Now, the functionality of the *ggplot2* package is available in our R session.

> Installing vs loading packages
>
> The main thing to remember is that you only need to install a package once, but you need to load it with library each time you wish to use it in a new R session. R will unload all of its packages each time you close RStudio.

4.1.3. Finding R packages

Finding useful R packages can be done in several ways. First, browsing CRAN (https://cran.r-project.org/) and Bioconductor (more later, https://bioconductor.org) are an excellent starting points, as they host thousands of packages categorized by topic. Additionally, online forums like Stack Overflow and R-bloggers can provide valuable recommendations based on user experiences. Social media platforms such as Twitter, where developers and data scientists often share new packages and updates, can also be a helpful resource. Finally, don't forget to ask your colleagues or fellow R users for their favorite packages, as they may have insights on which ones best suit your specific needs.

4.2. Are our dice fair?

Well, let's review our code.

```
roll2 <- function(bones = 1:6) {
  dice = sample(bones, size = 2, replace = TRUE)
  sum(dice)
}
```

If our dice are fair, then each number should show up equally. What does the sum look like with our two dice?

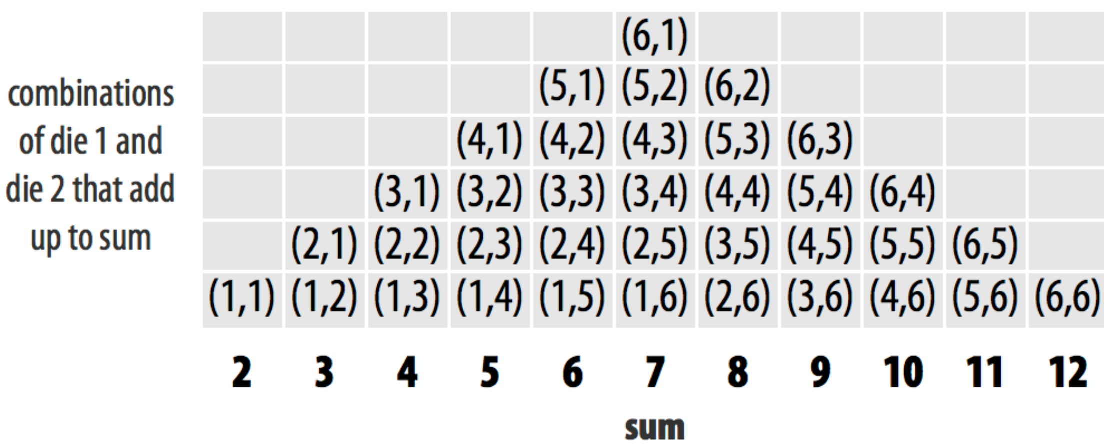


Figure 4.1.: In an ideal world, a histogram of the results would look like this

Read the help page for `replicate` (i.e., `help("replicate")`). In short, it suggests that we can repeat our dice rolling as many times as we like and replicate will return a *vector* of the sums for each roll.

```
rolls = replicate(n = 100, roll2())
```

What does `rolls` look like?

```
head(rolls)
```

```
[1] 7 5 6 6 3 4
```

```
length(rolls)
```

```
[1] 100
```

```
mean(rolls)
```

```
[1] 6.76
```

```
summary(rolls)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   3.00    5.00    7.00    6.76    9.00   12.00 
```

This looks like it roughly agrees with our sketched out ideal histogram in Figure 4.1. However, now that we've loaded the `qplot` function from the *ggplot2* package, we can make a histogram of the data themselves.

```
qplot(rolls, binwidth=1)
```

```
Warning: `qplot()` was deprecated in ggplot2 3.4.0.
```

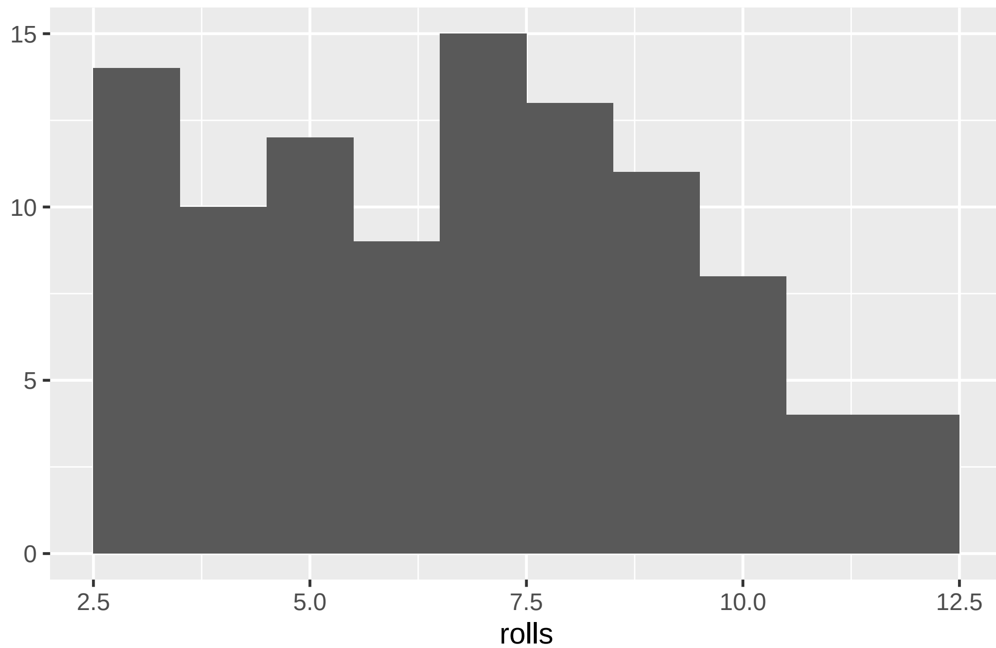


Figure 4.2.: Histogram of the sums from 100 rolls of our fair dice

How does your histogram look (and yours will be different from mine since we are sampling random values)? Is it what you expect?

What happens to our histogram as we increase the number of replicates?

```
rolls = replicate(n = 100000, roll2())
qplot(rolls, binwidth=1)
```

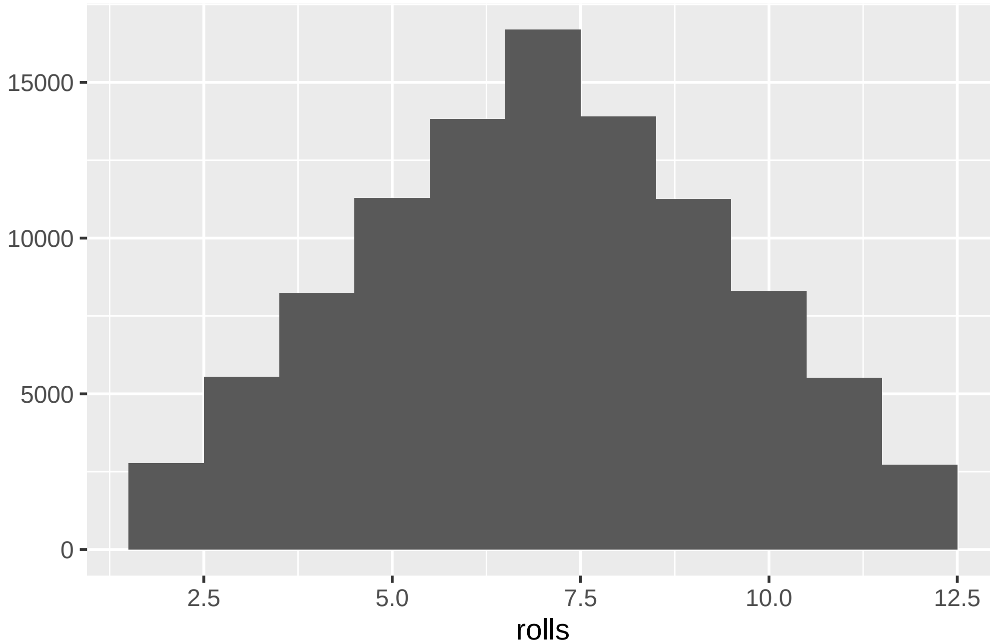


Figure 4.3.: Histogram with 100000 rolls much more closely approximates the pyramidal shape we anticipated

4.3. Bonus exercise

How would you change the `roll2` function to weight the dice?

5. Reading and writing data files

5.1. Introduction

In this chapter, we will discuss how to read and write data files in R. Data files are essential for storing and sharing data across different platforms and applications. R provides a variety of functions and packages to read and write data files in different formats, such as text files, CSV files, Excel files. By mastering these functions, you can efficiently import and export data in R, enabling you to perform data analysis and visualization tasks effectively.

5.2. CSV files

Comma-Separated Values (CSV) files are a common file format for storing tabular data. They consist of rows and columns, with each row representing a record and each column representing a variable or attribute. CSV files are widely used for data storage and exchange due to their simplicity and compatibility with various software applications. In R, you can read and write CSV files using the `read.csv()` and `write.csv()` functions, respectively. A commonly used alternative is to use the `readr` package, which provides faster and more user-friendly functions for reading and writing CSV files.

5.2.1. Writing a CSV file

Since we are going to use the `readr` package, we need to install it first. You can install the `readr` package using the following command:

```r
install.packages("readr")
```

Once the package is installed, you can load it into your R session using the `library()` function:

```
library(readr)
```

Since we don't have a CSV file sitting around, let's create a simple data frame to write to a CSV file. Here's an example data frame:

```
df <- data.frame(
  id = c(1, 2, 3, 4, 5),
  name = c("Alice", "Bob", "Charlie", "David", "Eve"),
  age = c(25, 30, 35, 40, 45)
)
```

Now, you can write this data frame to a CSV file using the `write_csv()` function from the `readr` package. Here's how you can do it:

```
write_csv(df, "data.csv")
```

You can check the current working directory to see if the CSV file was created successfully. If you want to specify a different directory or file path, you can provide the full path in the `write_csv()` function.

```
# see what the current working directory is
getwd()
```

```
[1] "/Users/seandavis/Documents/git/RBiocBook"
```

```
# and check to see that the file was created
dir(pattern = "data.csv")
```

```
[1] "data.csv"
```

5.2.2. Reading a CSV file

Now that we have a CSV file, let's read it back into R using the `read_csv()` function from the `readr` package. Here's how you can do it:

```
df2 <- read_csv("data.csv")
```

```
Rows: 5 Columns: 3
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): name
dbl (2): id, age

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

You can check the structure of the data frame `df2` to verify that the data was read correctly:

```
df2
```

```
# A tibble: 5 x 3
     id name      age
  <dbl> <chr>   <dbl>
1     1 Alice      25
2     2 Bob        30
3     3 Charlie    35
4     4 David      40
5     5 Eve        45
```

The `readr` package can read CSV files with various delimiters, headers, and data types, making it a versatile tool for handling tabular data in R. It can also read CSV files directly from web locations like so:

```
df3 <- read_csv("https://data.cdc.gov/resource/pwn4-m3yp.csv")
```

```
Rows: 1000 Columns: 10
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (1): state
dbl  (6): tot_cases, new_cases, tot_deaths, new_deaths, new_historic_cases, ...
dttm (3): date_updated, start_date, end_date

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

The dataset that you just downloaded is described here: Covid-19 data from CDC

5.3. Excel files

Microsoft Excel files are another common file format for storing tabular data. Excel files can contain multiple sheets, formulas, and formatting options, making them a popular choice for data storage and analysis. In R, you can read and write Excel files using the `readxl` package. This package provides functions to import and export data from Excel files, enabling you to work with Excel data in R.

5.3.1. Reading an Excel file

To read an Excel file in R, you need to install and load the `readxl` package. You can install the `readxl` package using the following command:

```
install.packages("readxl")
```

Once the package is installed, you can load it into your R session using the `library()` function:

```
library(readxl)
```

Now, you can read an Excel file using the `read_excel()` function from the `readxl` package. We don't have an excel file available, so let's download one from the internet. Here's an example:

```
download.file('https://www.w3resource.com/python-exercises/pandas/excel/SaleData.xlsx', 'Sa
```

Now, you can read the Excel file into R using the `read_excel()` function:

```
df_excel <- read_excel("SaleData.xlsx")
```

You can check the structure of the data frame `df_excel` to verify that the data was read correctly:

```r
df_excel
```

```
# A tibble: 45 x 8
   OrderDate           Region  Manager SalesMan  Item  Units Unit_price Sale_amt
   <dttm>              <chr>   <chr>   <chr>     <chr> <dbl>      <dbl>    <dbl>
 1 2018-01-06 00:00:00 East    Martha  Alexander Tele~    95       1198   113810
 2 2018-01-23 00:00:00 Central Hermann Shelli    Home~    50        500    25000
 3 2018-02-09 00:00:00 Central Hermann Luis      Tele~    36       1198    43128
 4 2018-02-26 00:00:00 Central Timothy David     Cell~    27        225     6075
 5 2018-03-15 00:00:00 West    Timothy Stephen   Tele~    56       1198    67088
 6 2018-04-01 00:00:00 East    Martha  Alexander Home~    60        500    30000
 7 2018-04-18 00:00:00 Central Martha  Steven    Tele~    75       1198    89850
 8 2018-05-05 00:00:00 Central Hermann Luis      Tele~    90       1198   107820
 9 2018-05-22 00:00:00 West    Douglas Michael   Tele~    32       1198    38336
10 2018-06-08 00:00:00 East    Martha  Alexander Home~    60        500    30000
# i 35 more rows
```

The `readxl` package provides various options to read Excel files with multiple sheets, specific ranges, and data types, making it a versatile tool for handling Excel data in R.

5.3.2. Writing an Excel file

To write an Excel file in R, you can use the `write_xlsx()` function from the `writexl` package. You can install the `writexl` package using the following command:

```r
install.packages("writexl")
```

Once the package is installed, you can load it into your R session using the `library()` function:

```r
library(writexl)
```

The `write_xlsx()` function allows you to write a data frame to an Excel file. Here's an example:

```r
write_xlsx(df, "data.xlsx")
```

You can check the current working directory to see if the Excel file was created successfully. If you want to specify a different directory or file path, you can provide the full path in the `write_xlsx()` function.

```
# see what the current working directory is
getwd()
```

```
[1] "/Users/seandavis/Documents/git/RBiocBook"
```

```
# and check to see that the file was created
dir(pattern = "data.xlsx")
```

```
[1] "data.xlsx"
```

5.4. Additional options

- Google Sheets: You can read and write data from Google Sheets using the `googlesheets4` package. This package provides functions to interact with Google Sheets, enabling you to import and export data from Google Sheets to R.
- JSON files: You can read and write JSON files using the `jsonlite` package. This package provides functions to convert R objects to JSON format and vice versa, enabling you to work with JSON data in R.
- Database files: You can read and write data from database files using the `DBI` and `RSQLite` packages. These packages provide functions to interact with various database systems, enabling you to import and export data from databases to R.

6. Plotting with ggplot2

The `ggplot2` package is a popular data visualization package in R. It is based on the Grammar of Graphics, a general scheme for data visualization that breaks up graphs into semantic components such as scales and layers. The Grammar of Graphics was developed by Leland Wilkinson in 1999 and is implemented in the `ggplot2` package by Hadley Wickham.

The Grammar of Graphics is a powerful framework for creating complex visualizations by combining simple components. Figure 6.1 illustrates the layered components of a data visualization, each contributing to the final plot. Each layer builds upon the previous one, though not all layers are required for every plot.
The `ggplot2` package provides a flexible and intuitive interface for creating a wide range of visualizations, from simple scatter plots to complex multi-layered plots.

This chapter provides an overview of the `ggplot2` package and its implementation of the Grammar of Graphics. We will cover the basic components of a `ggplot2` plot, including data, aesthetics, geometries, and themes.

6.1. Data

The first step in creating a `ggplot2` plot is to specify the data to be visualized. The data should be in a tidy format (Wickham (2014)), with each row representing an observation and each column representing a variable. The insurance dataset is described in the book Machine Learning with R by Brett Lantz. The dataset describes medical information and costs billed by health insurance companies for 1338 individuals in 2013, as compiled by the United States Census Bureau.

Variables include:

- **age** age of primary beneficiary
- **sex** insurance contractor gender, female, male

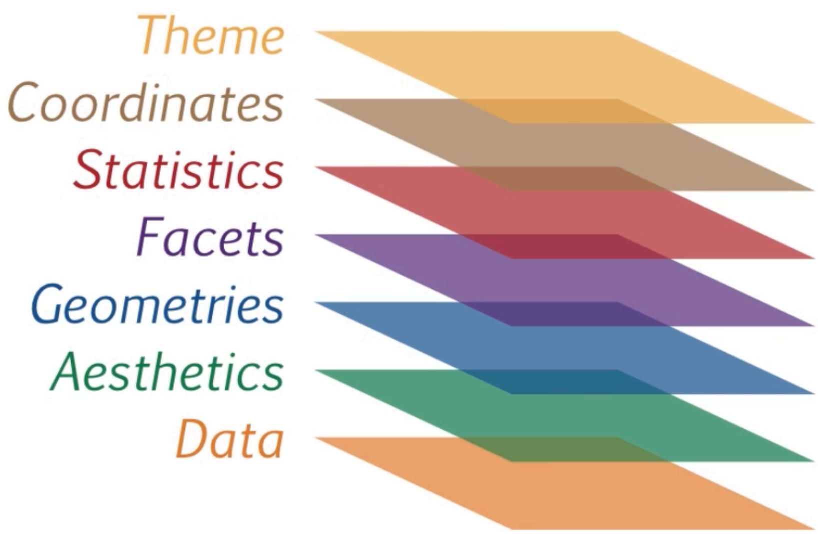


Figure 6.1.: Components of a Data Visualization Layer Structure. This diagram from Caron (2018) illustrates the layered components of a data visualization, each contributing to the final plot. Each layer builds upon the previous one, culminating in a comprehensive and interpretable visualization. Layers from bottom (foundation) to top (icing on the cake) are: 1) Data: The actual variables to be plotted. 2) Aesthetics: Scales onto which data is mapped. 3) Geometries: Shapes used to represent the data. 4) Facets: Rows and columns of sub-plots. 5) Statistics: Statistical models and summaries. 6) Coordinates: Plotting space for the data. 7) Theme: Describes all the non-data ink.

- **bmi** Body mass index, providing an understanding of body, weights that are relatively high or low relative to height, objective index of body weight (kg / m ^ 2) using the ratio of height to weight, ideally 18.5 to 24.9
- **children** Number of children covered by health insurance / Number of dependents
- **smoker** Smoking status
- **region** the beneficiary's residential area in the US, northeast, southeast, southwest, northwest.
- **charges** Individual medical costs billed by health insurance

We will load the data directly from the web, but you can also download the data from the link at github[1].

```
insurance_url <- "https://raw.githubusercontent.com/stedy/Machine-Learning-with-R-datasets/
insurance <- read.csv(insurance_url)
```

Explore the dataset a bit to understand its structure and contents. For example, you can use the `head()` function to view the first few rows of the dataset.

```
head(insurance)
```

```
  age    sex    bmi children smoker    region   charges
1  19 female 27.900        0    yes southwest 16884.924
2  18   male 33.770        1     no southeast  1725.552
3  28   male 33.000        3     no southeast  4449.462
4  33   male 22.705        0     no northwest 21984.471
5  32   male 28.880        0     no northwest  3866.855
6  31 female 25.740        0     no southeast  3756.622
```

And you can examine the dimensions of the dataset using the `dim()`, which returns the number of rows and columns in the dataset, the `ncol()` function, which returns the number of columns, and the `nrow()` function, which returns the number of rows.

```
dim(insurance)
```

```
[1] 1338    7
```

[1] Insurance data csv file, https://raw.githubusercontent.com/stedy/Machine-Learning-with-R-datasets/master/insurance.csv

```r
ncol(insurance)
```

```
[1] 7
```

```r
nrow(insurance)
```

```
[1] 1338
```

Note that with the `dim()` function, the number of rows is given first, followed by the number of columns.

Notice that, while the BMI variable represents a measure of a person's weight relative to their height, there is no discrete variable for whether a person is obese or not. The World Health Organization (WHO) defines obesity as a BMI greater than or equal to 30. We can create a new variable, `obese`, that indicates whether a person is obese based on their BMI.

```r
insurance$obese <- ifelse(insurance$bmi >= 30, "obese", "not obese")
```

If we examine the dataset again, we can see that the new variable `obese` has been added to the dataset.

```r
head(insurance)
```

```
  age    sex    bmi children smoker    region   charges     obese
1  19 female 27.900        0    yes southwest 16884.924 not obese
2  18   male 33.770        1     no southeast  1725.552     obese
3  28   male 33.000        3     no southeast  4449.462     obese
4  33   male 22.705        0     no northwest 21984.471 not obese
5  32   male 28.880        0     no northwest  3866.855 not obese
6  31 female 25.740        0     no southeast  3756.622 not obese
```

6.2. Aesthetics

The next step in creating a `ggplot2` plot is to specify the aesthetics of the plot. Aesthetics are visual properties of the plot that *map data to visual elements*.

```r
# specify dataset and mapping
library(ggplot2)
ggplot(
    data = insurance,
    mapping = aes(x = age, y = charges)
)
```

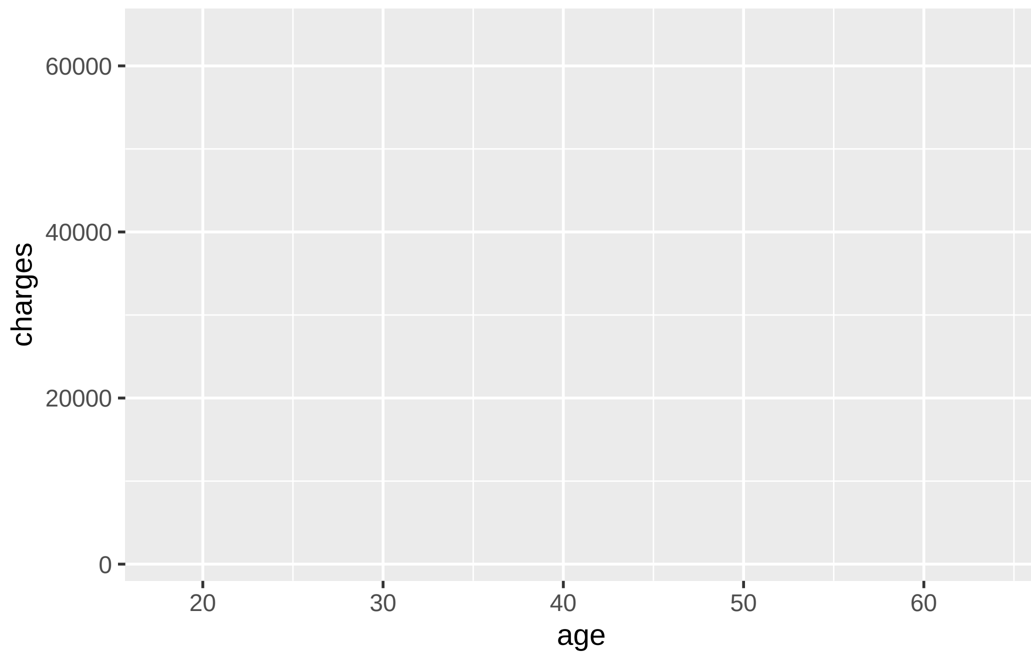


Figure 6.2.: A plot with age on the x-axis and charges on the y-axis.

In the code above, the `data` are the data to be visualized, and the `mapping` specifies how the data should be mapped to the plot. In this case, the `x` aesthetic is mapped to the `age` variable, and the `y` aesthetic is mapped to the `charges` variable. Note that there are no data displayed in Figure 6.2 yet; we have only specified the data and aesthetics. However, you can see the structure of the plot in the output, which shows the data and aesthetics that have been specified with `age` on the x-axis and `charges` on the y-axis.

6.3. Geometries

The next step is to add a geometry to the plot. Geometries are the visual representations of the data, such as points, lines, or bars. Since this is a scatter plot, we will use the `geom_point()` function to add points to the plot.

```
# add points to the plot
ggplot(
    data = insurance,
    mapping = aes(x = age, y = charges)
) +
    geom_point()
```

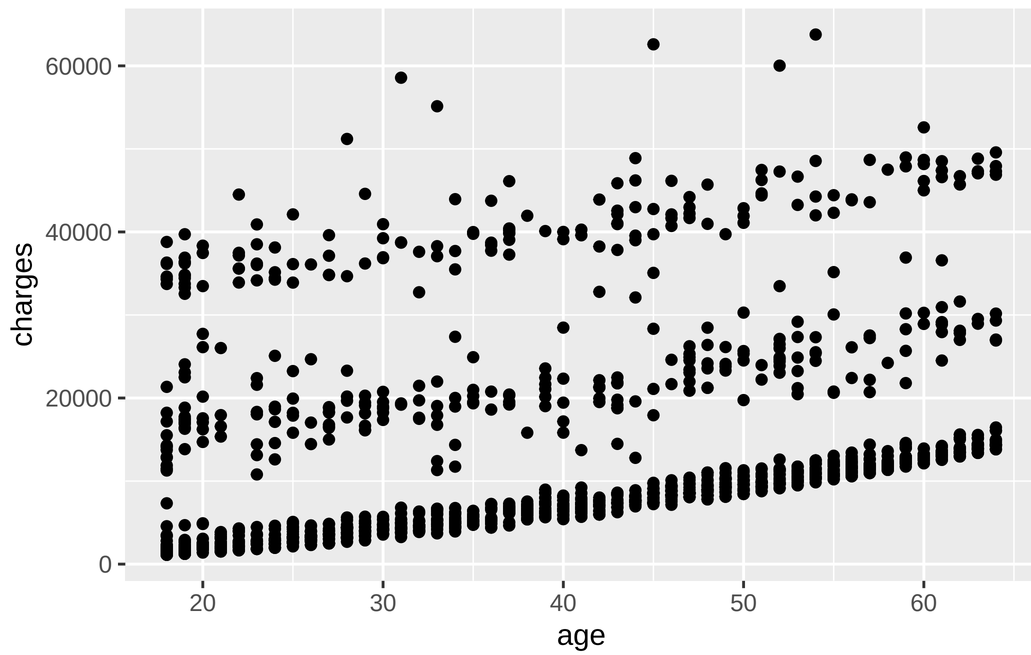


Figure 6.3.: A scatter plot with age on the x-axis and charges on the y-axis results from adding geom_point() to the plot.

> **ℹ Note**
>
> When using `ggplot2`, the `+` operator is used to add layers to the plot. The `ggplot()` function specifies the data and aesthetics, and the `geom_point()` function adds points to the plot. Using the `+` operator is a common practice in `ggplot2` to add layers to a plot, but the `+` operator does not work for other types of plots in R.

Using other geometries, you can create different types of plots. For example, you can use `geom_line()` to create a line plot, `geom_bar()` to create a bar plot, or `geom_boxplot()`

to create a box plot. Before doing so here, ask yourself if those geometries would be appropriate for the data you are working with.

A number of parameters (options) can be specified in a geom_ function. Options for the `geom_point()` function include color, size, and alpha. These control the point color, size, and transparency, respectively. Transparency ranges from 0 (completely transparent) to 1 (completely opaque). Adding a degree of transparency can help visualize overlapping points such as in Figure 6.4.

```
# add points to the plot
ggplot(
    data = insurance,
    mapping = aes(x = age, y = charges)
) +
    geom_point(
        color = "blue",
        size = 3,
        alpha = 0.3
    )
```

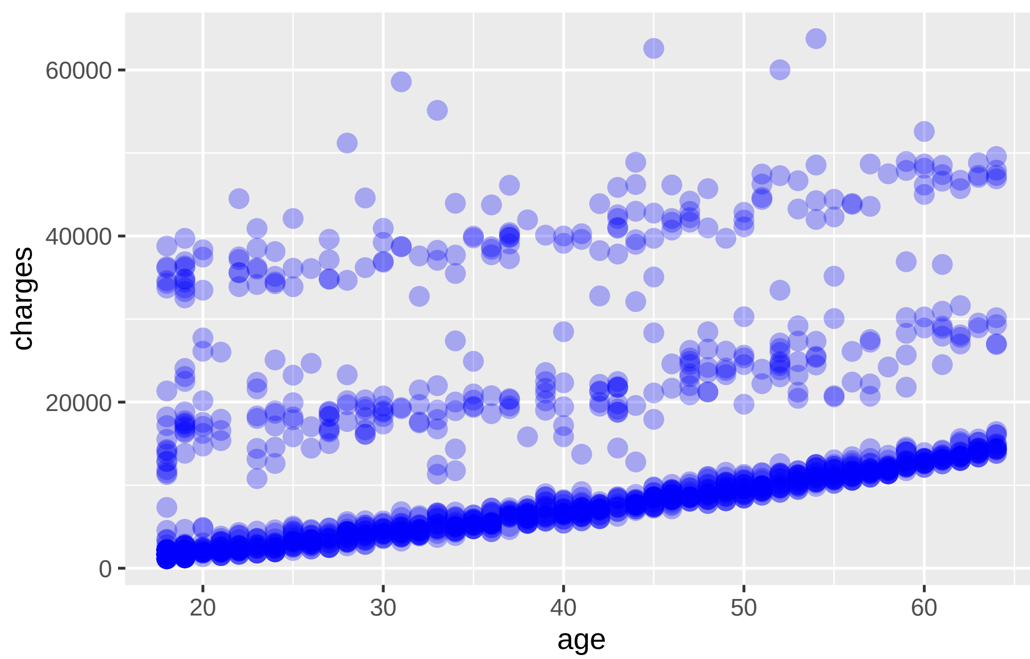


Figure 6.4.: A scatter plot with age on the x-axis and charges on the y-axis with colored points, larger size, and transparency.

We can add a best fit line to the scatter plot using the `geom_smooth()` function. The `method` parameter specifies the method used to fit the line. In this case, we will use the default method, which is linear regression, specified by `method = "lm"`. The `lm` method fits a *linear model* to the data, which in this case is simple linear regression [2] of the *dependent* variable `charges` as a function of the *independent* variable `age`. The result is shown in Figure 6.5.

```r
# add points and a best fit line to the plot
ggplot(
    data = insurance,
    mapping = aes(x = age, y = charges)
) +
    geom_point(
        color = "blue",
        alpha = 0.3
    ) +
    geom_smooth(method = "lm")
```

```
`geom_smooth()` using formula = 'y ~ x'
```

[2]The linear regression model is of the form $charges = \alpha + \beta * age + \epsilon$ where α is the intercept, β is the slope, and ϵ is the "error".

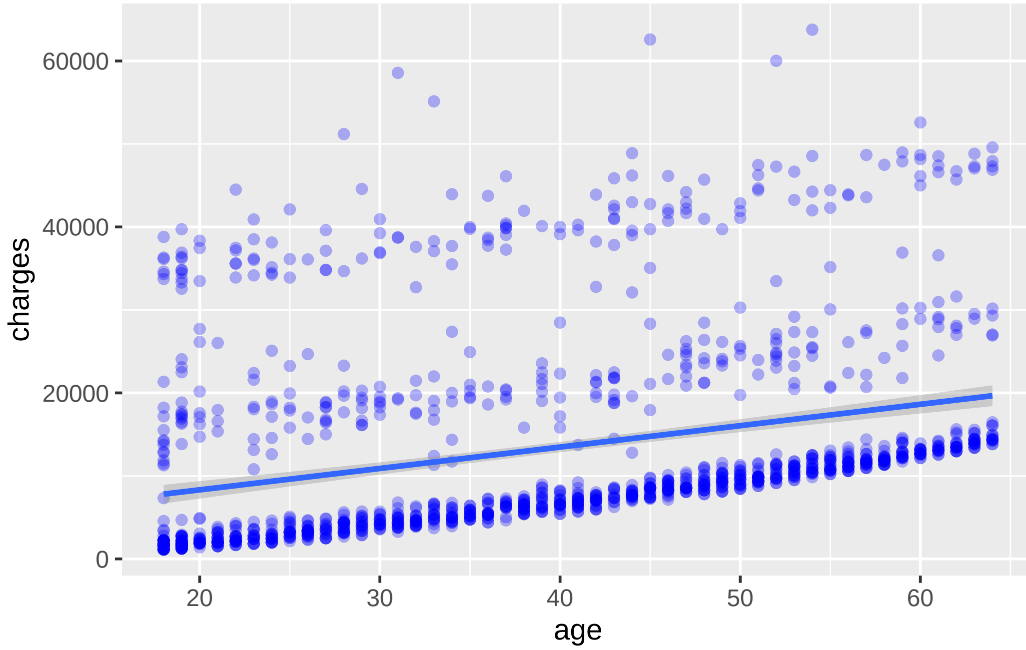


Figure 6.5.: A scatter plot with age on the x-axis and charges on the y-axis with a best fit line.

What do you observe in Figure 6.5 with the best fit line? How well does the line fit the data? Do you think a linear model is appropriate for this data?

6.4. Grouping

In addition to mapping variables to the x and y axes [i.e., `aes(x = ..., y=...)`], variables can be mapped to the color, shape, size, transparency, and other visual characteristics of geometric objects. This allows groups of observations to be superimposed in a single graph.

For example, we can map the `smoker` variable to the color of the points in the scatter plot. The result is shown in Figure 6.6.

```
# add points to the plot, colored by the smoker variable
ggplot(
    data = insurance,
    mapping = aes(x = age, y = charges, color = smoker)
```

```
) +
    geom_point()
```

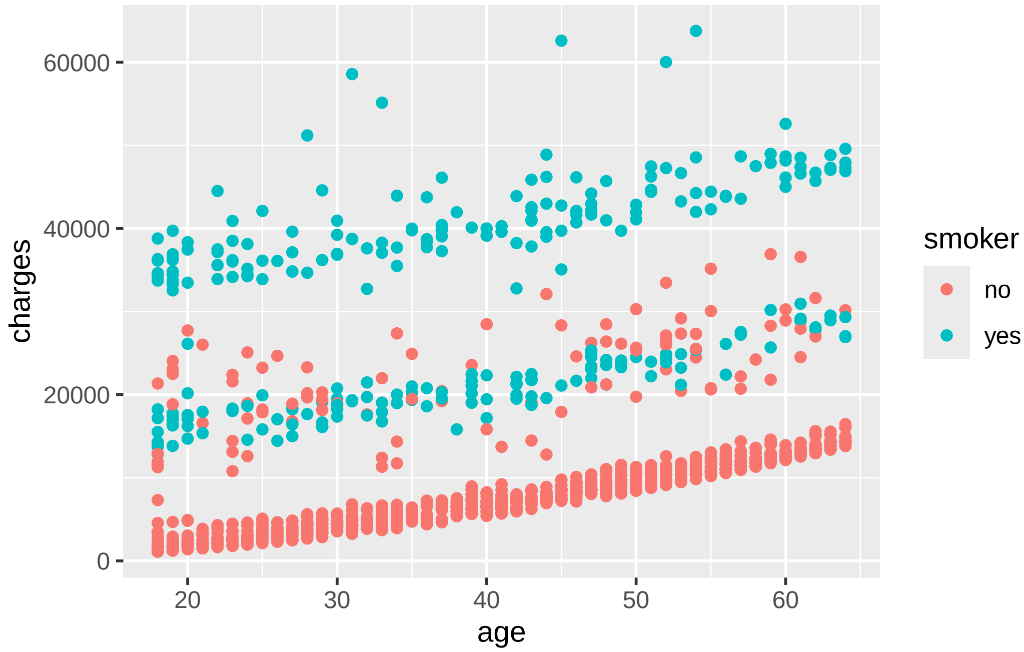


Figure 6.6.: A scatter plot with age on the x-axis and charges on the y-axis with points colored by the smoker variable.

In Figure 6.6, the points are colored based on the `smoker` variable, with smokers in orange and non-smokers in blue. This allows us to visually compare the charges of smokers and non-smokers as a function of age.

If we add back in the best fit line, we can see how the relationship between age and charges differs between smokers and non-smokers. The result is shown in Figure 6.7.

```
# add points to the plot, colored by the smoker variable, and a best fit line
ggplot(
    data = insurance,
    mapping = aes(x = age, y = charges, color = smoker)
) +
    geom_point() +
    geom_smooth(method = "lm")
```

```
`geom_smooth()` using formula = 'y ~ x'
```

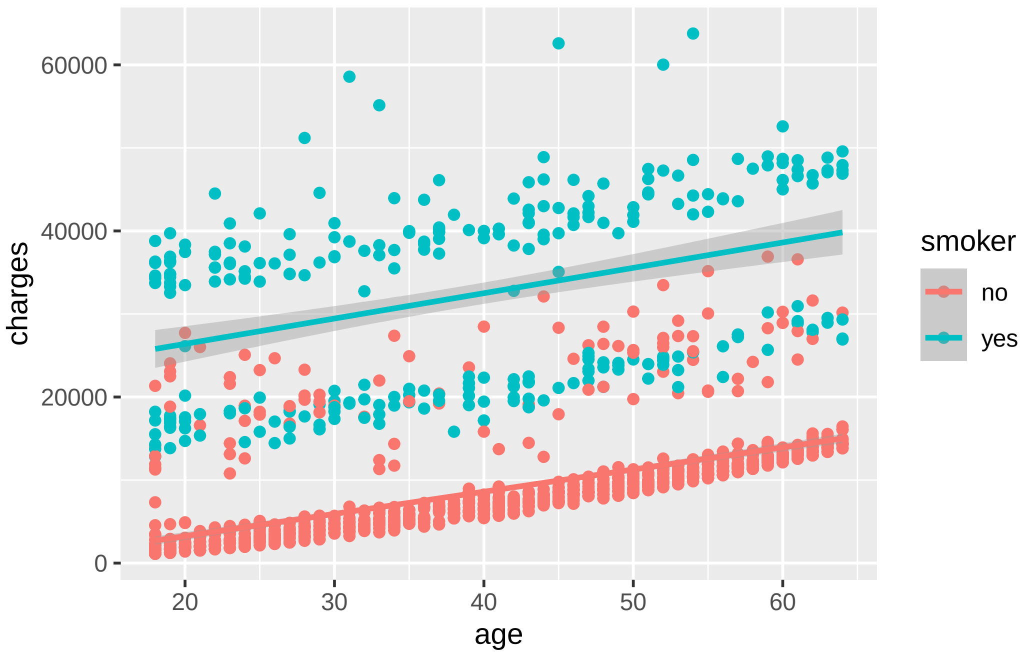


Figure 6.7.: A scatter plot with age on the x-axis and charges on the y-axis with points colored by the smoker variable and a best fit line.

How well does the best fit line fit the data for smokers and non-smokers? Do you see any differences in the relationship between age and charges for smokers and non-smokers?

6.5. Facets

Facets are a way to create multiple plots based on the levels of a categorical variable. In other words, facets allow you to create a grid of plots, with each plot showing a different subset of the data based on the levels of a categorical variable.

In Figure 6.7, we noticed that there are still two groups of points, even when looking at just smokers. We can further separate the data by the `obese` variable, creating a grid of plots with one plot for each combination of smoker and obese status.

```
# add points to the plot, colored by the smoker variable, and faceted by the obese variable
ggplot(
    data = insurance,
    mapping = aes(x = age, y = charges, color = smoker)
) +
    geom_point() +
    geom_smooth(method = "lm") +
    facet_wrap(~obese)
```

```
`geom_smooth()` using formula = 'y ~ x'
```

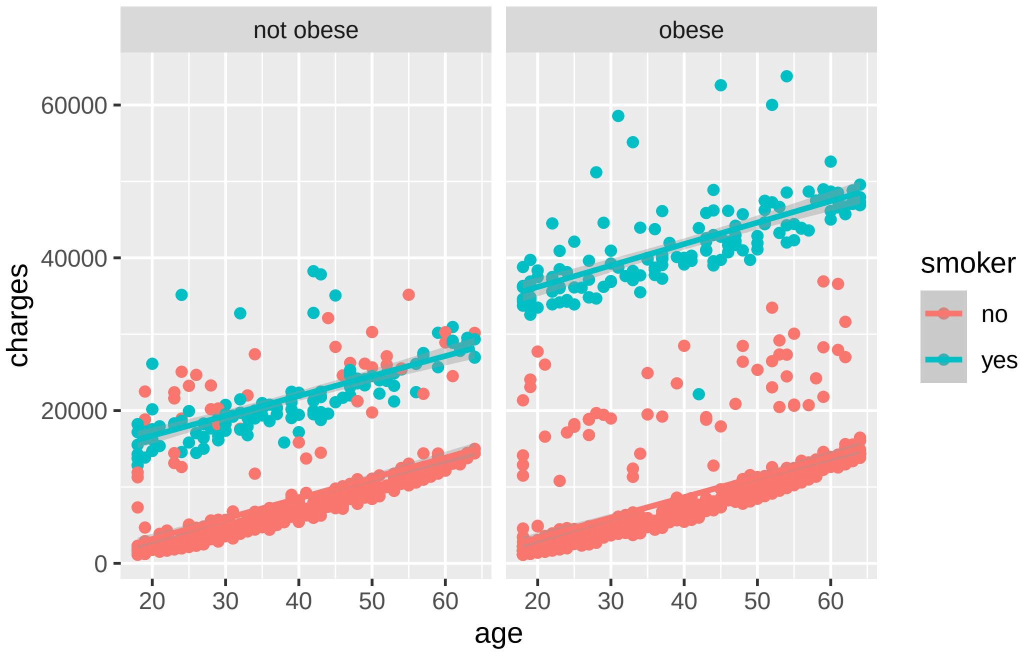


Figure 6.8.: A grid of scatter plots with age on the x-axis and charges on the y-axis, colored by the smoker variable, and faceted by the obese variable.

The way that we interpret the `facet_wrap(~ obese)` command is that we want to create a grid of plots, with each plot showing a different subset of the data based on the levels of the `obese` variable. In this case, we have two levels of the `obese` variable: `obese` and `not obese`, so we get two plots in the grid.

6.6. Labels

Labels are an important part of any plot. They help the viewer understand what the plot is showing and what the axes represent. While our plot already has labels for the x and y axes, we can add a title to the plot and change the labels for the x and y axes to make them more descriptive.

```
# add points to the plot, colored by the smoker variable, faceted by the obese variable, an
ggplot(
    data = insurance,
    mapping = aes(x = age, y = charges, color = smoker)
) +
    geom_point() +
    geom_smooth(method = "lm") +
    facet_wrap(~obese) +
    labs(
        title = "Medical Charges as a function of patient characteristics",
        subtitle = "US Census Bureau 2013 data",
        caption = "Source: https://github.com/stedy/Machine-Learning-with-R-datasets",
        x = "Age",
        y = "Annual Medical Charges",
        color = "Smoker?"
    )
```

```
`geom_smooth()` using formula = 'y ~ x'
```

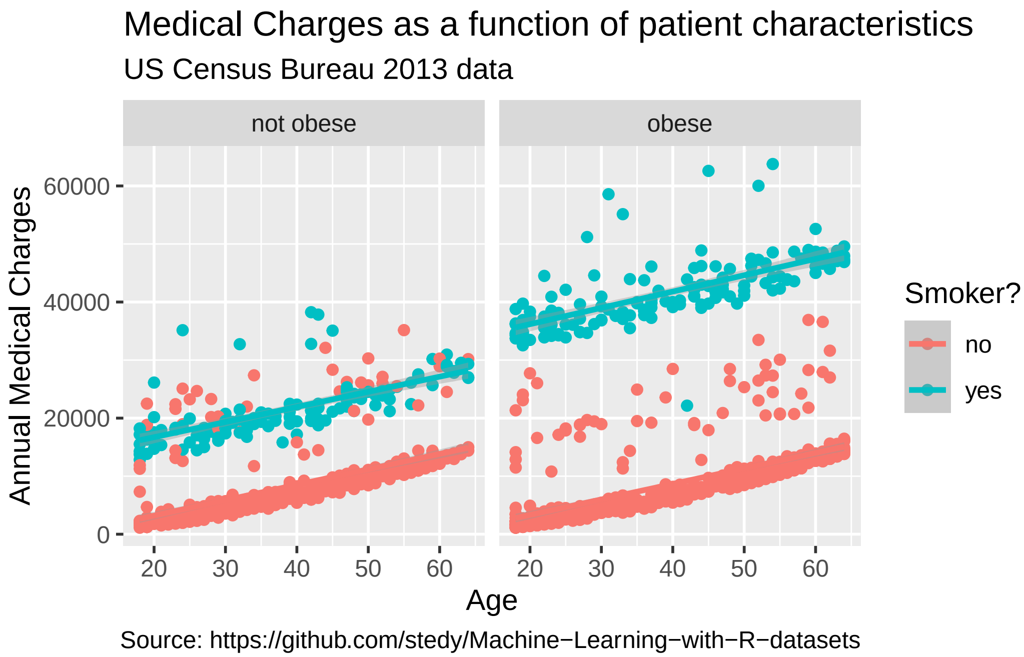


Figure 6.9.: A scatter plot with age on the x-axis and charges on the y-axis, colored by the smoker variable, and faceted by the obese variable, with labels.

6.7. Themes

Themes are a way to control the non-data ink in a plot, such as the background color, grid lines, and text size. Rather than specifying each element individually, you can use a pre-defined theme to quickly style your plot. For a nice overview of themes in `ggplot2`, see the the ggplot2 themes gallery.

To create a more visually appealing plot, we can apply the `theme_minimal()` theme to our plot. This theme removes the background grid lines and adds a light gray background to the plot.

```
# add points to the plot, colored by the smoker variable, faceted by the obese variable, ad
ggplot(
    data = insurance,
    mapping = aes(x = age, y = charges, color = smoker)
) +
    geom_point() +
    geom_smooth(method = "lm") +
```

```
    facet_wrap(~obese) +
    labs(
        title = "Medical Charges as a function of patient characteristics",
        subtitle = "US Census Bureau 2013 data",
        caption = "Source: https://github.com/stedy/Machine-Learning-with-R-datasets",
        x = "Age",
        y = "Annual Medical Charges",
        color = "Smoker?"
    ) +
    theme_minimal()
```

```
`geom_smooth()` using formula = 'y ~ x'
```

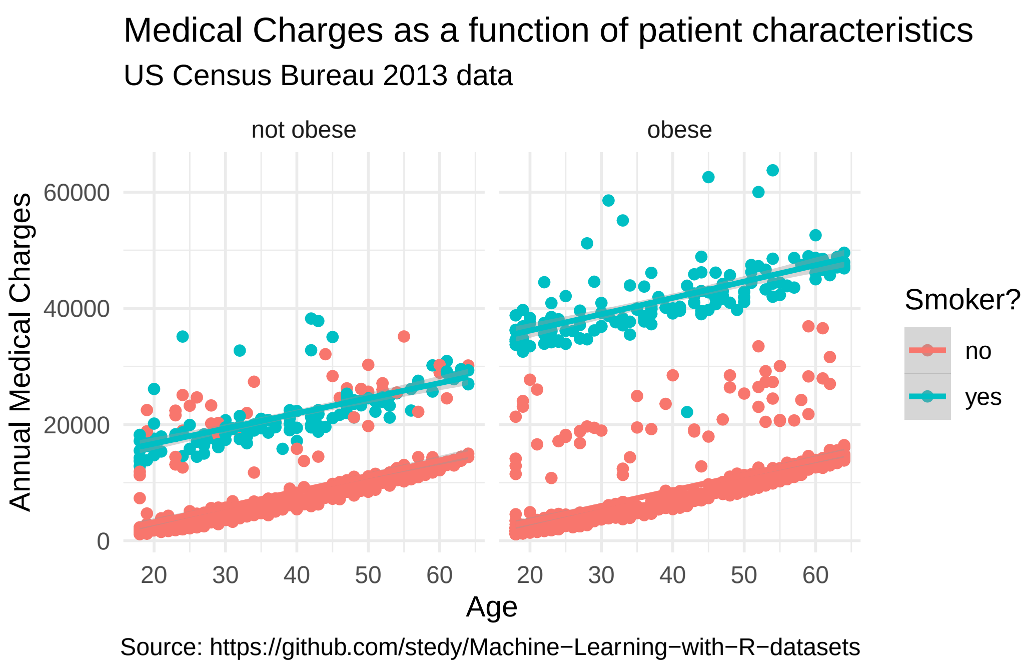


Figure 6.10.: A scatter plot with age on the x-axis and charges on the y-axis, colored by the smoker variable, faceted by the obese variable, with labels and a minimal theme.

6.8. Saving a Plot

Once you have created a plot that you are happy with, you may want to save it to a file for use in a report or presentation. The `ggsave()` function in `ggplot2` allows you to save a plot to a file in a variety of formats, including PNG, PDF, and SVG. Take a look at the help for `ggsave()` to see the available options. In particular, you can specify the file name, width, height, and resolution of the saved plot.

```r
# save the plot to a file
ggsave("insurance_plot.png")
```

```
Saving 5.5 x 3.5 in image
`geom_smooth()` using formula = 'y ~ x'
```

> **Note**
>
> The `ggsave()` function saves the last plot that you created with `ggplot2`. `ggsave()` will save the plot to the working directory by default, but you can specify a different directory by providing the full path to the file name.

References

Bourgon, Richard, Robert Gentleman, and Wolfgang Huber. 2010. "Independent Filtering Increases Detection Power for High-Throughput Experiments." *Proceedings of the National Academy of Sciences* 107 (21): 9546–51. https://doi.org/10.1073/pnas.0914005107.

Brouwer-Visser, Jurriaan, Wei-Yi Cheng, Anna Bauer-Mehren, Daniela Maisel, Katharina Lechner, Emilia Andersson, Joel T. Dudley, and Francesca Milletti. 2018. "Regulatory T-Cell Genes Drive Altered Immune Microenvironment in Adult Solid Cancers and Allow for Immune Contextual Patient Subtyping." *Cancer Epidemiology, Biomarkers & Prevention* 27 (1): 103–12. https://doi.org/10.1158/1055-9965.EPI-17-0461.

Buenrostro, Jason D, Paul G Giresi, Lisa C Zaba, Howard Y Chang, and William J Greenleaf. 2013. "Transposition of Native Chromatin for Fast and Sensitive Epigenomic Profiling of Open Chromatin, DNA-binding Proteins and Nucleosome Position." *Nature Methods* 10 (12): 1213–18. https://doi.org/10.1038/nmeth.2688.

Buenrostro, Jason D, Beijing Wu, Howard Y Chang, and William J Greenleaf. 2015. "ATAC-seq: A Method for Assaying Chromatin Accessibility Genome-Wide." *Current*

Protocols in Molecular Biology / Edited by Frederick M. Ausubel ... [Et Al.] 109 (January): 21.29.1–9. https://doi.org/10.1002/0471142727.mb2129s109.

Caron, Stéphane. 2018. "The Grammar of Graphics." https://dotlayer.org/en/grammar-of-graphics/.

Center, Pew Research. 2016. "Lifelong Learning and Technology." *Pew Research Center: Internet, Science & Tech*. https://www.pewresearch.org/internet/2016/03/22/lifelong-learning-and-technology/.

Crawford, Gregory E, Sean Davis, Peter C Scacheri, Gabriel Renaud, Mohamad J Halawi, Michael R Erdos, Roland Green, Paul S Meltzer, Tyra G Wolfsberg, and Francis S Collins. 2006. "DNase-chip: A High-Resolution Method to Identify DNase I Hypersensitive Sites Using Tiled Microarrays." *Nature Methods* 3 (7): 503–9. http://www.ncbi.nlm.nih.gov/pubmed/16791207?dopt=AbstractPlus.

Crawford, Gregory E, Ingeborg E Holt, James Whittle, Bryn D Webb, Denise Tai, Sean Davis, Elliott H Margulies, et al. 2006. "Genome-Wide Mapping of DNase Hypersensitive Sites Using Massively Parallel Signature Sequencing (MPSS)." *Genome Research* 16 (1): 123–31. http://www.ncbi.nlm.nih.gov/pubmed/16344561?dopt=AbstractPlus.

DeRisi, J. L., V. R. Iyer, and P. O. Brown. 1997. "Exploring the Metabolic and Genetic Control of Gene Expression on a Genomic Scale." *Science (New York, N.Y.)* 278 (5338): 680–86. https://doi.org/10.1126/science.278.5338.680.

Greener, Joe G., Shaun M. Kandathil, Lewis Moffat, and David T. Jones. 2022. "A Guide to Machine Learning for Biologists." *Nature Reviews Molecular Cell Biology* 23 (1): 40–55. https://doi.org/10.1038/s41580-021-00407-0.

Knowles, Malcolm S., Elwood F. Holton, and Richard A. Swanson. 2005. *The Adult Learner: The Definitive Classic in Adult Education and Human Resource Development*. 6th ed. Amsterdam ; Boston: Elsevier.

Lawrence, Michael, Wolfgang Huber, Hervé Pagès, Patrick Aboyoun, Marc Carlson, Robert Gentleman, Martin T Morgan, and Vincent J Carey. 2013. "Software for Computing and Annotating Genomic Ranges." *PLoS Computational Biology* 9 (8): e1003118. https://doi.org/10.1371/journal.pcbi.1003118.

Libbrecht, Maxwell W., and William Stafford Noble. 2015. "Machine Learning Applications in Genetics and Genomics." *Nature Reviews Genetics* 16 (6): 321–32. https://doi.org/10.1038/nrg3920.

Morgan, Martin, Herve Pages, V Obenchain, and N Hayden. 2016. "Rsamtools: Binary Alignment (BAM), FASTA, Variant Call (BCF), and Tabix File Import." *R Package Version* 1 (0): 677–89.

Student. 1908. "The Probable Error of a Mean." *Biometrika* 6 (1): 1–25. https://doi.org/10.2307/2331554.

Tsompana, Maria, and Michael J Buck. 2014. "Chromatin Accessibility: A Window into the Genome." *Epigenetics & Chromatin* 7 (1): 33. https://doi.org/10.1186/1756-8935-7-33.

Wickham, Hadley. 2014. "Tidy Data." *Journal of Statistical Software, Articles* 59 (10):

1–23. https://doi.org/10.18637/jss.v059.i10.

Part II.

R Data Structures

Welcome to the section on R data structures! As you begin your journey in learning R, it is essential to understand the fundamental building blocks of this powerful programming language. R offers a variety of data structures to store and manipulate data, each with its unique properties and capabilities. In this section, we will cover the core data structures in R, including:

- Vectors
- Matrices
- Lists
- Data.frames

By the end of this section, you will have a solid understanding of these data structures, and you will be able to choose and utilize the appropriate data structure for your specific data manipulation and analysis tasks.

In each chapter, we will delve into the properties and usage of each data structure, starting with their definitions and moving on to their practical applications. We will provide examples, exercises, and active learning approaches to help you better understand and apply these concepts in your work.

Chapter overview

- Vectors : In this chapter, we will introduce you to the simplest data structure in R, the vector. We will cover how to create, access, and manipulate vectors, as well as discuss their unique properties and limitations.

- **Matrices** Next, we will explore matrices, which are two-dimensional data structures that extend vectors. You will learn how to create, access, and manipulate matrices, and understand their usefulness in mathematical operations and data organization.

- **Lists** The third chapter will focus on lists, a versatile data structure that can store elements of different types and sizes. We will discuss how to create, access, and modify lists, and demonstrate their flexibility in handling complex data structures.

- **Data.frames** Finally, we will examine data.frames, a widely-used data structure for organizing and manipulating tabular data. You will learn how to create, access, and manipulate data.frames, and understand their advantages over other data structures for data analysis tasks.

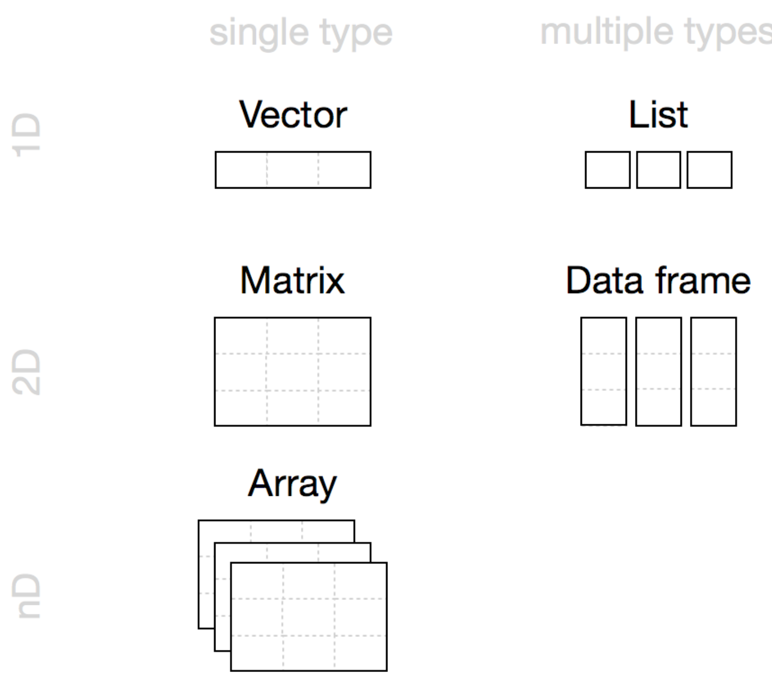


Figure 6.11.: A pictorial representation of R's most common data structures are vectors, matrices, arrays, lists, and dataframes. Figure from Hands-on Programming with R.

- **Arrays** While we will not focus directly on the `array` data type, which are multidimensional data structures that extend matrices, they are very similar to matrices, but with a third dimension.

As you progress through these chapters, practice the examples and exercises provided, engage in discussion, and collaborate with your peers to deepen your understanding of R data structures. This solid foundation will serve as the basis for more advanced data manipulation, analysis, and visualization techniques in R.

7. Vectors

7.1. What is a Vector?

A vector is the simplest and most basic data structure in R. It is a one-dimensional, ordered collection of elements, where all the elements are of the same data type. Vectors can store various types of data, such as numeric, character, or logical values. Figure 7.1 shows a pictorial representation of three vector examples.

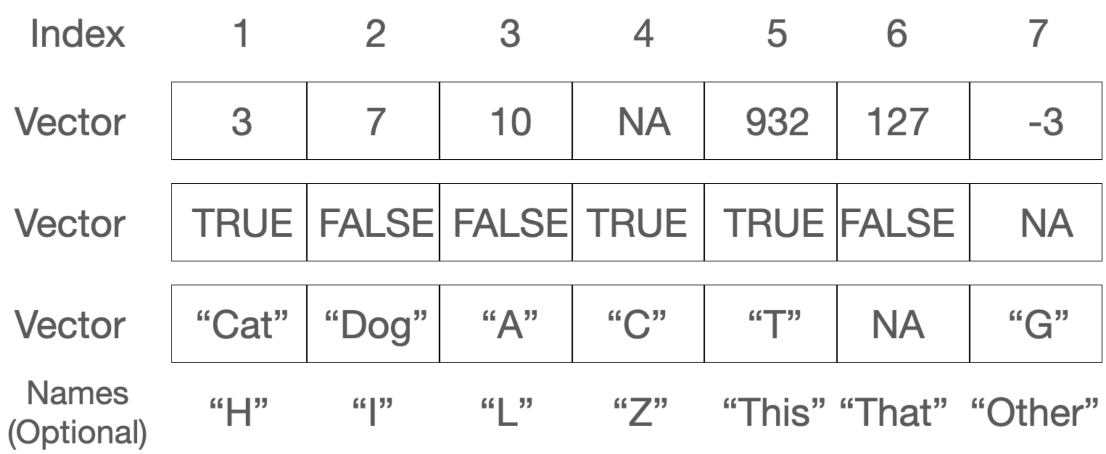


Figure 7.1.: “Pictorial representation of three vector examples. The first vector is a numeric vector. The second is a ‘logical’ vector. The third is a character vector. Vectors also have indices and, optionally, names.”

In this chapter, we will provide a comprehensive overview of vectors, including how to create, access, and manipulate them. We will also discuss some unique properties and rules associated with vectors, and explore their applications in data analysis tasks.

In R, even a single value is a vector with length=1.

```r
z = 1
z
```

```
[1] 1
```

```r
length(z)
```

```
[1] 1
```

In the code above, we "assigned" the value 1 to the variable named `z`. Typing `z` by itself is an "expression" that returns a result which is, in this case, the value that we just assigned. The `length` method takes an R object and returns the R length. There are numerous ways of asking R about what an object represents, and `length` is one of them.

Vectors can contain numbers, strings (character data), or logical values (`TRUE` and `FALSE`) or other "atomic" data types Table 7.1. *Vectors cannot contain a mix of types!* We will introduce another data structure, the R `list` for situations when we need to store a mix of base R data types.

Table 7.1.: Atomic (simplest) data types in R.

Data type	Stores
numeric	floating point numbers
integer	integers
complex	complex numbers
factor	categorical data
character	strings
logical	TRUE or FALSE
NA	missing
NULL	empty
function	function type

7.2. Creating vectors

Character vectors (also sometimes called "string" vectors) are entered with each value surrounded by single or double quotes; either is acceptable, but they must match. They are always displayed by R with double quotes. Here are some examples of creating vectors:

```
# examples of vectors
c('hello','world')
```

```
[1] "hello" "world"
```

```
c(1,3,4,5,1,2)
```

```
[1] 1 3 4 5 1 2
```

```
c(1.12341e7,78234.126)
```

```
[1] 11234100.00    78234.13
```

```
c(TRUE,FALSE,TRUE,TRUE)
```

```
[1]  TRUE FALSE  TRUE  TRUE
```

```
# note how in the next case the TRUE is converted to "TRUE"
# with quotes around it.
c(TRUE,'hello')
```

```
[1] "TRUE"  "hello"
```

We can also create vectors as "regular sequences" of numbers. For example:

```
# create a vector of integers from 1 to 10
x = 1:10
# and backwards
x = 10:1
```

The `seq` function can create more flexible regular sequences.

```
# create a vector of numbers from 1 to 4 skipping by 0.3
y = seq(1,4,0.3)
```

And creating a new vector by concatenating existing vectors is possible, as well.

```
# create a sequence by concatenating two other sequences
z = c(y,x)
z
```

```
 [1]  1.0  1.3  1.6  1.9  2.2  2.5  2.8  3.1  3.4  3.7  4.0 10.0  9.0  8.0  7.0
[16]  6.0  5.0  4.0  3.0  2.0  1.0
```

7.3. Vector Operations

Operations on a single vector are typically done element-by-element. For example, we can add `2` to a vector, `2` is added to each element of the vector and a new vector of the same length is returned.

```
x = 1:10
x + 2
```

```
 [1]  3  4  5  6  7  8  9 10 11 12
```

If the operation involves two vectors, the following rules apply. If the vectors are the same length: R simply applies the operation to each pair of elements.

```
x + x
```

```
 [1]  2  4  6  8 10 12 14 16 18 20
```

If the vectors are different lengths, but one length a multiple of the other, R reuses the shorter vector as needed.

```
x = 1:10
y = c(1,2)
x * y
```

```
 [1]  1  4  3  8  5 12  7 16  9 20
```

If the vectors are different lengths, but one length *not* a multiple of the other, R reuses the shorter vector as needed *and* delivers a warning.

```
x = 1:10
y = c(2,3,4)
x * y
```

```
Warning in x * y: longer object length is not a multiple of shorter object
length
```

```
 [1]  2  6 12  8 15 24 14 24 36 20
```

Typical operations include multiplication ("*"), addition, subtraction, division, exponentiation ("^"), but many operations in R operate on vectors and are then called "vectorized".

Be aware of the recycling rule when working with vectors of different lengths, as it may lead to unexpected results if you're not careful.

7.4. Logical Vectors

Logical vectors are vectors composed on only the values `TRUE` and `FALSE`. Note the all-upper-case and no quotation marks.

```
a = c(TRUE,FALSE,TRUE)

# we can also create a logical vector from a numeric vector
# 0 = false, everything else is 1
b = c(1,0,217)
d = as.logical(b)
d
```

```
[1]  TRUE FALSE  TRUE
```

```
# test if a and d are the same at every element
all.equal(a,d)
```

```
[1] TRUE
```

```
# We can also convert from logical to numeric
as.numeric(a)
```

```
[1] 1 0 1
```

7.4.1. Logical Operators

Some operators like `<, >, ==, >=, <=, !=` can be used to create logical vectors.

```
# create a numeric vector
x = 1:10
# testing whether x > 5 creates a logical vector
x > 5
```

```
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
x <= 5
```

```
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
```

```
x != 5
```

```
 [1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
x == 5
```

```
 [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
```

We can also assign the results to a variable:

```
y = (x == 5)
y
```

```
 [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
```

7.5. Indexing Vectors

In R, an index is used to refer to a specific element or set of elements in an vector (or other data structure). [R uses `[` and `]` to perform indexing, although other approaches to getting subsets of larger data structures are common in R.

```
x = seq(0,1,0.1)
# create a new vector from the 4th element of x
x[4]
```

```
[1] 0.3
```

We can even use other vectors to perform the "indexing".

```
x[c(3,5,6)]
```

```
[1] 0.2 0.4 0.5
```

```
y = 3:6
x[y]
```

```
[1] 0.2 0.3 0.4 0.5
```

Combining the concept of indexing with the concept of logical vectors results in a very power combination.

```
# use help('rnorm') to figure out what is happening next
myvec = rnorm(10)

# create logical vector that is TRUE where myvec is >0.25
gt1 = (myvec > 0.25)
sum(gt1)
```

```
[1] 4
```

```
# and use our logical vector to create a vector of myvec values that are >0.25
myvec[gt1]
```

```
[1] 1.1484509 1.1463211 0.7716711 0.2969809
```

```
# or <=0.25 using the logical "not" operator, "!"
myvec[!gt1]
```

```
[1] -0.4014349 -0.5081373 -0.4925580 -1.6429488 -0.1851662 -1.0668761
```

```
# shorter, one line approach
myvec[myvec > 0.25]
```

```
[1] 1.1484509 1.1463211 0.7716711 0.2969809
```

7.6. Named Vectors

Named vectors are vectors with labels or names assigned to their elements. These names can be used to access and manipulate the elements in a more meaningful way.

To create a named vector, use the `names()` function:

```
fruit_prices <- c(0.5, 0.75, 1.25)
names(fruit_prices) <- c("apple", "banana", "cherry")
print(fruit_prices)
```

```
 apple banana cherry 
  0.50   0.75   1.25 
```

You can also access and modify elements using their names:

```
banana_price <- fruit_prices["banana"]
print(banana_price)
```

```
banana 
  0.75 
```

```
fruit_prices["apple"] <- 0.6
print(fruit_prices)
```

```
 apple banana cherry
  0.60   0.75   1.25
```

7.7. Character Vectors, A.K.A. Strings

R uses the `paste` function to concatenate strings.

```
paste("abc","def")
```

```
[1] "abc def"
```

```
paste("abc","def",sep="THISSEP")
```

```
[1] "abcTHISSEPdef"
```

```
paste0("abc","def")
```

```
[1] "abcdef"
```

```
## [1] "abcdef"
paste(c("X","Y"),1:10)
```

```
 [1] "X 1"  "Y 2"  "X 3"  "Y 4"  "X 5"  "Y 6"  "X 7"  "Y 8"  "X 9"  "Y 10"
```

```
paste(c("X","Y"),1:10,sep="_")
```

```
 [1] "X_1"  "Y_2"  "X_3"  "Y_4"  "X_5"  "Y_6"  "X_7"  "Y_8"  "X_9"  "Y_10"
```

We can count the number of characters in a string.

```r
nchar('abc')
```

```
[1] 3
```

```r
nchar(c('abc','d',123456))
```

```
[1] 3 1 6
```

Pulling out parts of strings is also sometimes useful.

```r
substr('This is a good sentence.',start=10,stop=15)
```

```
[1] " good "
```

Another common operation is to replace something in a string with something (a find-and-replace).

```r
sub('This','That','This is a good sentence.')
```

```
[1] "That is a good sentence."
```

When we want to find all strings that match some other string, we can use `grep`, or "grab regular expression".

```r
grep('bcd',c('abcdef','abcd','bcde','cdef','defg'))
```

```
[1] 1 2 3
```

```r
grep('bcd',c('abcdef','abcd','bcde','cdef','defg'),value=TRUE)
```

```
[1] "abcdef" "abcd"   "bcde"
```

Read about the `grepl` function (`?grepl`). Use that function to return a logical vector (TRUE/FALSE) for each entry above with an `a` in it.

7.8. Missing Values, AKA "NA"

R has a special value, "NA", that represents a "missing" value, or *Not Available*, in a vector or other data structure. Here, we just create a vector to experiment.

```
x = 1:5
x
```

```
[1] 1 2 3 4 5
```

```
length(x)
```

```
[1] 5
```

```
is.na(x)
```

```
[1] FALSE FALSE FALSE FALSE FALSE
```

```
x[2] = NA
x
```

```
[1]  1 NA  3  4  5
```

The length of `x` is unchanged, but there is one value that is marked as "missing" by virtue of being `NA`.

```
length(x)
```

```
[1] 5
```

```
is.na(x)
```

```
[1] FALSE  TRUE FALSE FALSE FALSE
```

We can remove `NA` values by using indexing. In the following, `is.na(x)` returns a logical vector the length of `x`. The `!` is the logical *NOT* operator and converts `TRUE` to `FALSE` and vice-versa.

```
x[!is.na(x)]
```

```
[1] 1 3 4 5
```

7.9. Exercises

1. Create a numeric vector called `temperatures` containing the following values: 72, 75, 78, 81, 76, 73.

   ```
   temperatures <- c(72, 75, 78, 81, 76, 73, 93)
   ```

2. Create a character vector called `days` containing the following values: "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday".

   ```
   days <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"
   ```

3. Calculate the average temperature for the week and store it in a variable called `average_temperature`.

   ```
   average_temperature <- mean(temperatures)
   ```

4. Create a named vector called `weekly_temperatures`, where the names are the days of the week and the values are the temperatures from the `temperatures` vector.

   ```
   weekly_temperatures <- temperatures
   names(weekly_temperatures) <- days
   ```

5. Create a numeric vector called `ages` containing the following values: 25, 30, 35, 40, 45, 50, 55, 60.

   ```
   ages <- c(25, 30, 35, 40, 45, 50, 55, 60)
   ```

6. Create a logical vector called `is_adult` by checking if the elements in the `ages` vector are greater than or equal to 18.

   ```
   is_adult <- ages >= 18
   ```

7. Calculate the sum and product of the `ages` vector.

   ```
   sum_ages <- sum(ages)
   product_ages <- prod(ages)
   ```

8. Extract the ages greater than or equal to 40 from the `ages` vector and store them in a variable called `older_ages`.

```
older_ages <- ages[ages >= 40]
```

8. Matrices

A *matrix* is a rectangular collection of the same data type (see Figure 8.1). It can be viewed as a collection of column vectors all of the same length and the same type (i.e. numeric, character or logical) OR a collection of row vectors, again all of the same type and length. A *data.frame* is *also* a rectangular array. All of the columns must be the same length, but they **may be** of *different* types. The rows and columns of a matrix or data frame can be given names. However these are implemented differently in R; many operations will work for one but not both, often a source of confusion.

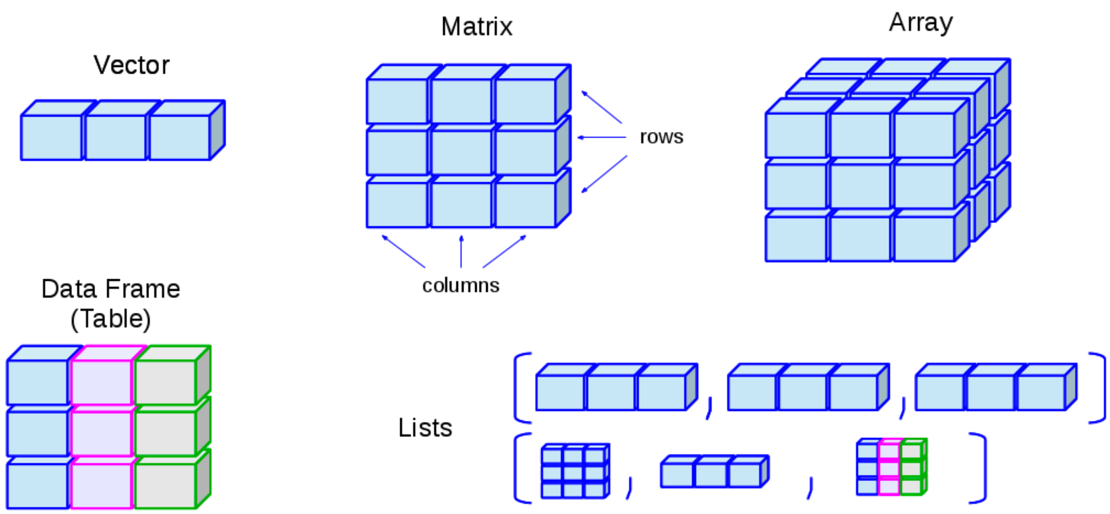


Figure 8.1.: A matrix is a collection of column vectors.

8.1. Creating a matrix

There are many ways to create a matrix in R. One of the simplest is to use the `matrix()` function. In the code below, we'll create a matrix from a vector from 1:16.

```
mat1 <- matrix(1:16,nrow=4)
mat1
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
```

The same is possible, but specifying that the matrix be "filled" by row.

```
mat1 <- matrix(1:16,nrow=4,byrow = TRUE)
mat1
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
[4,]   13   14   15   16
```

Notice the subtle difference in the order that the numbers go into the matrix.

We can also build a matrix from parts by "binding" vectors together:

```
x <- 1:10
y <- rnorm(10)
```

Each of the vectors above is of length 10 and both are "numeric", so we can make them into a matrix. Using `rbind` binds rows (**r**) into a matrix.

```
mat <- rbind(x,y)
mat
```

```
        [,1]      [,2]      [,3]      [,4]      [,5]      [,6]       [,7]
x  1.0000000 2.0000000 3.0000000 4.0000000 5.0000000  6.000000  7.0000000
y -0.1007675 0.5519366 0.4488688 0.3981466 0.8524107 -1.027999 -0.6854053
       [,8]       [,9]      [,10]
x 8.0000000  9.0000000 10.0000000
y 0.4897315 -0.2333974  0.7278752
```

The alternative to `rbind` is `cbind` that binds columns (**c**) together.

```
mat <- cbind(x,y)
mat
```

```
       x          y
 [1,]  1 -0.1007675
 [2,]  2  0.5519366
 [3,]  3  0.4488688
 [4,]  4  0.3981466
 [5,]  5  0.8524107
 [6,]  6 -1.0279989
 [7,]  7 -0.6854053
 [8,]  8  0.4897315
 [9,]  9 -0.2333974
[10,] 10  0.7278752
```

Inspecting the names associated with rows and columns is often useful, particularly if the names have human meaning.

```
rownames(mat)
```

```
NULL
```

```
colnames(mat)
```

```
[1] "x" "y"
```

We can also change the names of the matrix by assigning *valid* names to the columns or rows.

```
colnames(mat) = c('apples','oranges')
colnames(mat)
```

```
[1] "apples"  "oranges"
```

```r
mat
```

```
      apples    oranges
 [1,]      1 -0.1007675
 [2,]      2  0.5519366
 [3,]      3  0.4488688
 [4,]      4  0.3981466
 [5,]      5  0.8524107
 [6,]      6 -1.0279989
 [7,]      7 -0.6854053
 [8,]      8  0.4897315
 [9,]      9 -0.2333974
[10,]     10  0.7278752
```

Matrices have dimensions.

```r
dim(mat)
```

```
[1] 10  2
```

```r
nrow(mat)
```

```
[1] 10
```

```r
ncol(mat)
```

```
[1] 2
```

8.2. Accessing elements of a matrix

Indexing for matrices works as for vectors except that we now need to include both the row and column (in that order). We can access elements of a matrix using the square bracket [indexing method. Elements can be accessed as `var[r, c]`. Here, `r` and `c` are vectors describing the elements of the matrix to select.

> **! Important**
>
> The indices in R start with one, meaning that the first element of a vector or the first row/column of a matrix is indexed as one.
> This is different from some other programming languages, such as Python, which use zero-based indexing, meaning that the first element of a vector or the first row/column of a matrix is indexed as zero.
> It is important to be aware of this difference when working with data in R, especially if you are coming from a programming background that uses zero-based indexing. Using the wrong index can lead to unexpected results or errors in your code.

```r
# The 2nd element of the 1st row of mat
mat[1,2]
```

```
   oranges
-0.1007675
```

```r
# The first ROW of mat
mat[1,]
```

```
    apples    oranges
 1.0000000 -0.1007675
```

```r
# The first COLUMN of mat
mat[,1]
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```r
# and all elements of mat that are > 4; note no comma
mat[mat>4]
```

```
[1]  5  6  7  8  9 10
```

```
## [1]  5  6  7  8  9 10
```

> Caution
>
> Note that in the last case, there is no ",", so R treats the matrix as a long vector (length=20). This is convenient, sometimes, but it can also be a source of error, as some code may "work" but be doing something unexpected.

We can also use indexing to exclude a row or column by prefixing the selection with a `-` sign.

```
mat[,-1]       # remove first column
```

```
 [1] -0.1007675  0.5519366  0.4488688  0.3981466  0.8524107 -1.0279989
 [7] -0.6854053  0.4897315 -0.2333974  0.7278752
```

```
mat[-c(1:5),]  # remove first five rows
```

```
     apples    oranges
[1,]      6 -1.0279989
[2,]      7 -0.6854053
[3,]      8  0.4897315
[4,]      9 -0.2333974
[5,]     10  0.7278752
```

8.3. Changing values in a matrix

We can create a matrix filled with random values drawn from a normal distribution for our work below.

```
m = matrix(rnorm(20),nrow=10)
summary(m)
```

```
       V1                 V2          
 Min.   :-2.1707   Min.   :-1.78021  
 1st Qu.:-1.4913   1st Qu.:-0.68510  
 Median :-0.1734   Median :-0.37670  
 Mean   :-0.1912   Mean   :-0.04895  
 3rd Qu.: 0.5173   3rd Qu.: 0.96281  
 Max.   : 2.6163   Max.   : 1.39484  
```

Multiplication and division works similarly to vectors. When multiplying by a vector, for example, the values of the vector are reused. In the simplest case, let's multiply the matrix by a constant (vector of length 1).

```
# multiply all values in the matrix by 20
m2 = m*20
summary(m2)
```

```
       V1                V2
 Min.   :-43.414   Min.   :-35.604
 1st Qu.:-29.826   1st Qu.:-13.702
 Median : -3.467   Median : -7.534
 Mean   : -3.823   Mean   : -0.979
 3rd Qu.: 10.347   3rd Qu.: 19.256
 Max.   : 52.326   Max.   : 27.897
```

By combining subsetting with assignment, we can make changes to just part of a matrix.

```
# and add 100 to the first column of m
m2[,1] = m2[,1] + 100
# summarize m
summary(m2)
```

```
       V1               V2
 Min.   : 56.59   Min.   :-35.604
 1st Qu.: 70.17   1st Qu.:-13.702
 Median : 96.53   Median : -7.534
 Mean   : 96.18   Mean   : -0.979
 3rd Qu.:110.35   3rd Qu.: 19.256
 Max.   :152.33   Max.   : 27.897
```

A somewhat common transformation for a matrix is to transpose which changes rows to columns. One might need to do this if an assay output from a lab machine puts samples in rows and genes in columns, for example, while in Bioconductor/R, we often want the samples in columns and the genes in rows.

```
t(m2)
```

```
          [,1]       [,2]      [,3]     [,4]      [,5]      [,6]      [,7]
[1,] 76.265132 130.080007  68.14308 67.34310 152.32616 106.24387  56.58636
[2,] -5.879369  -9.188765 -18.58693 27.89672  20.71269 -35.60416 -13.89722
          [,8]     [,9]    [,10]
[1,] 111.71414 102.8434 90.22191
[2,]  14.88633 -13.1163 22.98715
```

8.4. Calculations on matrix rows and columns

Again, we just need a matrix to play with. We'll use `rnorm` again, but with a slight twist.

```
m3 = matrix(rnorm(100,5,2),ncol=10) # what does the 5 mean here? And the 2?
```

Since these data are from a normal distribution, we can look at a row (or column) to see what the mean and standard deviation are.

```
mean(m3[,1])
```

```
[1] 5.434771
```

```
sd(m3[,1])
```

```
[1] 1.675129
```

```
# or a row
mean(m3[1,])
```

```
[1] 6.147223
```

```
sd(m3[1,])
```

```
[1] 1.630307
```

There are some useful convenience functions for computing means and sums of data in **all** of the columns and rows of matrices.

```r
colMeans(m3)
```

```
 [1] 5.434771 5.177531 5.179380 4.965027 4.933516 4.238210 5.186793 3.976971
 [9] 4.788226 4.295322
```

```r
rowMeans(m3)
```

```
 [1] 6.147223 3.438289 4.920728 5.254608 3.609042 5.730218 4.280746 4.563036
 [9] 5.325723 4.906131
```

```r
rowSums(m3)
```

```
 [1] 61.47223 34.38289 49.20728 52.54608 36.09042 57.30218 42.80746 45.63036
 [9] 53.25723 49.06131
```

```r
colSums(m3)
```

```
 [1] 54.34771 51.77531 51.79380 49.65027 49.33516 42.38210 51.86793 39.76971
 [9] 47.88226 42.95322
```

We can look at the distribution of column means:

```r
# save as a variable
cmeans = colMeans(m3)
summary(cmeans)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  3.977   4.419   4.949   4.818   5.179   5.435 
```

Note that this is centered pretty closely around the selected mean of `5` above.

How about the standard deviation? There is not a `colSd` function, but it turns out that we can easily apply functions that take vectors as input, like `sd` and "apply" them across either the rows (the first dimension) or columns (the second) dimension.

```
csds = apply(m3, 2, sd)
summary(csds)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  1.054   1.677   1.791   1.811   1.953   2.420 
```

Again, take a look at the distribution which is centered quite close to the selected standard deviation when we created our matrix.

8.5. Exercises

8.5.1. Data preparation

For this set of exercises, we are going to rely on a dataset that comes with R. It gives the number of sunspots per month from 1749-1983. The dataset comes as a `ts` or time series data type which I convert to a matrix using the following code.

Just run the code as is and focus on the rest of the exercises.

```
data(sunspots)
sunspot_mat <- matrix(as.vector(sunspots),ncol=12,byrow = TRUE)
colnames(sunspot_mat) <- as.character(1:12)
rownames(sunspot_mat) <- as.character(1749:1983)
```

8.5.2. Questions

- After the conversion above, what does `sunspot_mat` look like? Use functions to find the number of rows, the number of columns, the class, and some basic summary statistics.

  ```
  ncol(sunspot_mat)
  nrow(sunspot_mat)
  dim(sunspot_mat)
  summary(sunspot_mat)
  head(sunspot_mat)
  tail(sunspot_mat)
  ```

- Practice subsetting the matrix a bit by selecting:

- The first 10 years (rows)
- The month of July (7th column)
- The value for July, 1979 using the rowname to do the selection.

```
sunspot_mat[1:10,]
sunspot_mat[,7]
sunspot_mat['1979',7]
```

1. These next few exercises take advantage of the fact that calling a univariate statistical function (one that expects a vector) works for matrices by just making a vector of all the values in the matrix. What is the highest (max) number of sunspots recorded in these data?

```
max(sunspot_mat)
```

2. And the minimum?

```
min(sunspot_mat)
```

3. And the overall mean and median?

```
mean(sunspot_mat)
median(sunspot_mat)
```

4. Use the `hist()` function to look at the distribution of all the monthly sunspot data.

```
hist(sunspot_mat)
```

5. Read about the `breaks` argument to `hist()` to try to increase the number of breaks in the histogram to increase the resolution slightly. Adjust your `hist()` and breaks to your liking.

```
hist(sunspot_mat, breaks=40)
```

6. Now, let's move on to summarizing the data a bit to learn about the pattern of sunspots varies by month or by year. Examine the dataset again. What do the columns represent? And the rows?

```
# just a quick glimpse of the data will give us a sense
head(sunspot_mat)
```

7. We'd like to look at the distribution of sunspots by month. How can we do that?

```
# the mean of the columns is the mean number of sunspots per month.
colMeans(sunspot_mat)
```

```
# Another way to write the same thing:
apply(sunspot_mat, 2, mean)
```

8. Assign the month summary above to a variable and summarize it to get a sense of the spread over months.

```
monthmeans = colMeans(sunspot_mat)
summary(monthmeans)
```

9. Play the same game for years to get the per-year mean?

```
ymeans = rowMeans(sunspot_mat)
summary(ymeans)
```

10. Make a plot of the yearly means. Do you see a pattern?

```
plot(ymeans)
# or make it clearer
plot(ymeans, type='l')
```

9. Data Frames

While R has many different data types, the one that is central to much of the power and popularity of R is the `data.frame`. A data.frame looks a bit like an R matrix in that it has two dimensions, rows and columns. However, data.frames are usually viewed as a set of columns representing variables and the rows representing the values of those variables. Importantly, a data.frame may contain *different* data types in each of its columns; matrices **must** contain only one data type. This distinction is important to remember, as there are *specific* approaches to working with R data.frames that may be different than those for working with matrices.

9.1. Learning goals

- Understand how data.frames are different from matrices.
- Know a few functions for examing the contents of a data.frame.
- List approaches for subsetting data.frames.
- Be able to load and save tabular data from and to disk.
- Show how to create a data.frames from scratch.

9.2. Learning objectives

- Load the yeast growth dataset into R using `read.csv`.
- Examine the contents of the dataset.
- Use subsetting to find genes that may be involved with nutrient metabolism and transport.
- Summarize data measurements by categories.

9.3. Dataset

The data used here are borrowed directly from the fantastic Bioconnector tutorials and are a cleaned up version of the data from Brauer et al. Coordination of Growth Rate, Cell

Cycle, Stress Response, and Metabolic Activity in Yeast (2008) Mol Biol Cell 19:352-367. These data are from a gene expression microarray, and in this paper the authors examine the relationship between growth rate and gene expression in yeast cultures limited by one of six different nutrients (glucose, leucine, ammonium, sulfate, phosphate, uracil). If you give yeast a rich media loaded with nutrients except restrict the supply of a single nutrient, you can control the growth rate to any rate you choose. By starving yeast of specific nutrients you can find genes that:

1. Raise or lower their expression in response to growth rate. Growth-rate dependent expression patterns can tell us a lot about cell cycle control, and how the cell responds to stress. The authors found that expression of >25% of all yeast genes is linearly correlated with growth rate, independent of the limiting nutrient. They also found that the subset of negatively growth-correlated genes is enriched for peroxisomal functions, and positively correlated genes mainly encode ribosomal functions.
2. Respond differently when different nutrients are being limited. If you see particular genes that respond very differently when a nutrient is sharply restricted, these genes might be involved in the transport or metabolism of that specific nutrient.

The dataset can be downloaded directly from:

- brauer2007_tidy.csv

We are going to read this dataset into R and then use it as a playground for learning about data.frames.

9.4. Reading in data

R has many capabilities for reading in data. Many of the functions have names that help us to understand what data format is to be expected. In this case, the filename that we want to read ends in `.csv`, meaning comma-separated-values. The `read.csv()` function reads in `.csv` files. As usual, it is worth reading `help('read.csv')` to get a better sense of the possible bells-and-whistles.

The `read.csv()` function can read directly from a URL, so we do not need to download the file directly. This dataset is relatively large (about 16MB), so this may take a bit depending on your network connection speed.

```r
options(width=60)
```

```
url = paste0(
    'https://raw.githubusercontent.com',
    '/bioconnector/workshops/master/data/brauer2007_tidy.csv'
)
ydat <- read.csv(url)
```

Our variable, `ydat`, now "contains" the downloaded and read data. We can check to see what data type read.csv gave us:

```
class(ydat)
```

```
[1] "data.frame"
```

9.5. Inspecting data.frames

Our `ydat` variable is a data.frame. As I mentioned, the dataset is fairly large, so we will not be able to look at it all at once on the screen. However, R gives us many tools to inspect a data.frame.

- Overviews of content
  - `head()` to show first few rows
  - `tail()` to show last few rows
- Size
  - `dim()` for dimensions (rows, columns)
  - `nrow()`
  - `ncol()`
  - `object.size()` for power users interested in the memory used to store an object
- Data and attribute summaries
  - `colnames()` to get the names of the columns
  - `rownames()` to get the "names" of the rows–may not be present
  - `summary()` to get per-column summaries of the data in the data.frame.

```
head(ydat)
```

```
  symbol systematic_name nutrient rate expression
1   SFB2         YNL049C  Glucose 0.05      -0.24
2   <NA>         YNL095C  Glucose 0.05       0.28
3   QRI7         YDL104C  Glucose 0.05      -0.02
4   CFT2         YLR115W  Glucose 0.05      -0.33
5   SSO2         YMR183C  Glucose 0.05       0.05
6   PSP2         YML017W  Glucose 0.05      -0.69
                            bp
1        ER to Golgi transport
2   biological process unknown
3 proteolysis and peptidolysis
4      mRNA polyadenylylation*
5              vesicle fusion*
6   biological process unknown
                             mf
1    molecular function unknown
2    molecular function unknown
3 metalloendopeptidase activity
4                   RNA binding
5              t-SNARE activity
6    molecular function unknown
```

```r
tail(ydat)
```

```
       symbol systematic_name nutrient rate expression
198425   DOA1         YKL213C   Uracil  0.3       0.14
198426   KRE1         YNL322C   Uracil  0.3       0.28
198427   MTL1         YGR023W   Uracil  0.3       0.27
198428   KRE9         YJL174W   Uracil  0.3       0.43
198429   UTH1         YKR042W   Uracil  0.3       0.19
198430   <NA>         YOL111C   Uracil  0.3       0.04
                                             bp
198425    ubiquitin-dependent protein catabolism*
198426      cell wall organization and biogenesis
198427      cell wall organization and biogenesis
198428     cell wall organization and biogenesis*
198429 mitochondrion organization and biogenesis*
198430                 biological process unknown
                                 mf
198425         molecular function unknown
```

```
198426 structural constituent of cell wall
198427          molecular function unknown
198428          molecular function unknown
198429          molecular function unknown
198430          molecular function unknown
```

```r
dim(ydat)
```

```
[1] 198430      7
```

```r
nrow(ydat)
```

```
[1] 198430
```

```r
ncol(ydat)
```

```
[1] 7
```

```r
colnames(ydat)
```

```
[1] "symbol"          "systematic_name" "nutrient"
[4] "rate"            "expression"      "bp"
[7] "mf"
```

```r
summary(ydat)
```

```
    symbol           systematic_name      nutrient
 Length:198430      Length:198430      Length:198430
 Class :character   Class :character   Class :character
 Mode  :character   Mode  :character   Mode  :character



      rate          expression             bp
 Min.   :0.0500   Min.   :-6.500000   Length:198430
 1st Qu.:0.1000   1st Qu.:-0.290000   Class :character
```

```
 Median :0.2000   Median : 0.000000   Mode  :character  
 Mean   :0.1752   Mean   : 0.003367                     
 3rd Qu.:0.2500   3rd Qu.: 0.290000                     
 Max.   :0.3000   Max.   : 6.640000                     
      mf           
 Length:198430     
 Class :character  
 Mode  :character  
```

In RStudio, there is an additional function, `View()` (note the capital "V") that opens the first 1000 rows (default) in the RStudio window, akin to a spreadsheet view.

```r
View(ydat)
```

9.6. Accessing variables (columns) and subsetting

In R, data.frames can be subset similarly to other two-dimensional data structures. The `[` in R is used to denote subsetting of any kind. When working with two-dimensional data, we need two values inside the `[ ]` to specify the details. The specification is `[rows, columns]`. For example, to get the first three rows of `ydat`, use:

```r
ydat[1:3, ]
```

```
  symbol systematic_name nutrient rate expression
1   SFB2         YNL049C  Glucose 0.05      -0.24
2   <NA>         YNL095C  Glucose 0.05       0.28
3   QRI7         YDL104C  Glucose 0.05      -0.02
                            bp
1        ER to Golgi transport
2   biological process unknown
3 proteolysis and peptidolysis
                             mf
1    molecular function unknown
2    molecular function unknown
3 metalloendopeptidase activity
```

Note how the second number, the columns, is blank. R takes that to mean “all the columns”. Similarly, we can combine rows and columns specification arbitrarily.

```
ydat[1:3, 1:3]
```

```
  symbol systematic_name nutrient
1   SFB2         YNL049C  Glucose
2   <NA>         YNL095C  Glucose
3   QRI7         YDL104C  Glucose
```

Because selecting a single variable, or column, is such a common operation, there are two shortcuts for doing so *with data.frames*. The first, the `$` operator works like so:

```
# Look at the column names, just to refresh memory
colnames(ydat)
```

```
[1] "symbol"          "systematic_name" "nutrient"
[4] "rate"            "expression"      "bp"
[7] "mf"
```

```
# Note that I am using "head" here to limit the output
head(ydat$symbol)
```

```
[1] "SFB2" NA     "QRI7" "CFT2" "SSO2" "PSP2"
```

```
# What is the actual length of "symbol"?
length(ydat$symbol)
```

```
[1] 198430
```

The second is related to the fact that, in R, data.frames are also lists. We subset a list by using `[[]]` notation. To get the second column of `ydat`, we can use:

```
head(ydat[[2]])
```

```
[1] "YNL049C" "YNL095C" "YDL104C" "YLR115W" "YMR183C"
[6] "YML017W"
```

Alternatively, we can use the column name:

```
head(ydat[["systematic_name"]])
```

```
[1] "YNL049C" "YNL095C" "YDL104C" "YLR115W" "YMR183C"
[6] "YML017W"
```

9.6.1. Some data exploration

There are a couple of columns that include numeric values. Which columns are numeric?

```
class(ydat$symbol)
```

```
[1] "character"
```

```
class(ydat$rate)
```

```
[1] "numeric"
```

```
class(ydat$expression)
```

```
[1] "numeric"
```

Make histograms of: - the expression values - the rate values

What does the `table()` function do? Could you use that to look a the `rate` column given that that column appears to have repeated values?

What `rate` corresponds to the most nutrient-starved condition?

9.6.2. More advanced indexing and subsetting

We can use, for example, logical values (TRUE/FALSE) to subset data.frames.

```
head(ydat[ydat$symbol == 'LEU1', ])
```

```
     symbol systematic_name nutrient rate expression   bp
NA     <NA>            <NA>     <NA>   NA         NA <NA>
NA.1   <NA>            <NA>     <NA>   NA         NA <NA>
NA.2   <NA>            <NA>     <NA>   NA         NA <NA>
NA.3   <NA>            <NA>     <NA>   NA         NA <NA>
NA.4   <NA>            <NA>     <NA>   NA         NA <NA>
NA.5   <NA>            <NA>     <NA>   NA         NA <NA>
       mf
NA   <NA>
NA.1 <NA>
NA.2 <NA>
NA.3 <NA>
NA.4 <NA>
NA.5 <NA>
```

```
tail(ydat[ydat$symbol == 'LEU1', ])
```

```
         symbol systematic_name nutrient rate expression
NA.47244   <NA>            <NA>     <NA>   NA         NA
NA.47245   <NA>            <NA>     <NA>   NA         NA
NA.47246   <NA>            <NA>     <NA>   NA         NA
NA.47247   <NA>            <NA>     <NA>   NA         NA
NA.47248   <NA>            <NA>     <NA>   NA         NA
NA.47249   <NA>            <NA>     <NA>   NA         NA
           bp   mf
NA.47244 <NA> <NA>
NA.47245 <NA> <NA>
NA.47246 <NA> <NA>
NA.47247 <NA> <NA>
NA.47248 <NA> <NA>
NA.47249 <NA> <NA>
```

What is the problem with this approach? It appears that there are a bunch of `NA` values. Taking a quick look at the `symbol` column, we see what the problem.

```r
summary(ydat$symbol)
```

```
   Length     Class      Mode 
   198430 character character 
```

Using the `is.na()` function, we can make filter further to get down to values of interest.

```r
head(ydat[ydat$symbol == 'LEU1' & !is.na(ydat$symbol), ])
```

```
      symbol systematic_name nutrient rate expression
1526    LEU1         YGL009C  Glucose 0.05      -1.12
7043    LEU1         YGL009C  Glucose 0.10      -0.77
12555   LEU1         YGL009C  Glucose 0.15      -0.67
18071   LEU1         YGL009C  Glucose 0.20      -0.59
23603   LEU1         YGL009C  Glucose 0.25      -0.20
29136   LEU1         YGL009C  Glucose 0.30       0.03
                         bp
1526  leucine biosynthesis
7043  leucine biosynthesis
12555 leucine biosynthesis
18071 leucine biosynthesis
23603 leucine biosynthesis
29136 leucine biosynthesis
                                          mf
1526  3-isopropylmalate dehydratase activity
7043  3-isopropylmalate dehydratase activity
12555 3-isopropylmalate dehydratase activity
18071 3-isopropylmalate dehydratase activity
23603 3-isopropylmalate dehydratase activity
29136 3-isopropylmalate dehydratase activity
```

Sometimes, looking at the data themselves is not that important. Using `dim()` is one possibility to look at the number of rows and columns after subsetting.

```r
dim(ydat[ydat$expression > 3, ])
```

```
[1] 714   7
```

Find the high expressed genes when leucine-starved. For this task we can also use `subset` which allows us to treat column names as R variables (no `$` needed).

```r
subset(ydat, nutrient == 'Leucine' & rate == 0.05 & expression > 3)
```

```
       symbol systematic_name nutrient rate expression
133768   QDR2         YIL121W  Leucine 0.05       4.61
133772   LEU1         YGL009C  Leucine 0.05       3.84
133858   BAP3         YDR046C  Leucine 0.05       4.29
135186   <NA>         YPL033C  Leucine 0.05       3.43
135187   <NA>         YLR267W  Leucine 0.05       3.23
135288   HXT3         YDR345C  Leucine 0.05       5.16
135963   TPO2         YGR138C  Leucine 0.05       3.75
135965   YRO2         YBR054W  Leucine 0.05       4.40
136102   GPG1         YGL121C  Leucine 0.05       3.08
136109  HSP42         YDR171W  Leucine 0.05       3.07
136119   HXT5         YHR096C  Leucine 0.05       4.90
136151   <NA>         YJL144W  Leucine 0.05       3.06
136152   MOH1         YBL049W  Leucine 0.05       3.43
136153   <NA>         YBL048W  Leucine 0.05       3.95
136189  HSP26         YBR072W  Leucine 0.05       4.86
136231   NCA3         YJL116C  Leucine 0.05       4.03
136233   <NA>         YBR116C  Leucine 0.05       3.28
136486   <NA>         YGR043C  Leucine 0.05       3.07
137443   ADH2         YMR303C  Leucine 0.05       4.15
137448   ICL1         YER065C  Leucine 0.05       3.54
137451   SFC1         YJR095W  Leucine 0.05       3.72
137569   MLS1         YNL117W  Leucine 0.05       3.76
                                             bp
133768                        multidrug transport
133772                      leucine biosynthesis
133858                      amino acid transport
135186                                  meiosis*
135187                biological process unknown
135288                          hexose transport
135963                       polyamine transport
135965                biological process unknown
136102                       signal transduction
136109                        response to stress*
136119                          hexose transport
```

```
136151                   response to dessication
136152                biological process unknown
136153                                      <NA>
136189                       response to stress*
136231 mitochondrion organization and biogenesis
136233                                      <NA>
136486                biological process unknown
137443                             fermentation*
137448                          glyoxylate cycle
137451                       fumarate transport*
137569                          glyoxylate cycle
                                         mf
133768         multidrug efflux pump activity
133772 3-isopropylmalate dehydratase activity
133858        amino acid transporter activity
135186            molecular function unknown
135187            molecular function unknown
135288          glucose transporter activity*
135963          spermine transporter activity
135965            molecular function unknown
136102              signal transducer activity
136109              unfolded protein binding
136119          glucose transporter activity*
136151            molecular function unknown
136152            molecular function unknown
136153                                   <NA>
136189              unfolded protein binding
136231            molecular function unknown
136233                                   <NA>
136486                transaldolase activity
137443         alcohol dehydrogenase activity
137448              isocitrate lyase activity
137451 succinate:fumarate antiporter activity
137569               malate synthase activity
```

9.7. Aggregating data

Aggregating data, or summarizing by category, is a common way to look for trends or differences in measurements between categories. Use `aggregate` to find the mean expression by gene symbol.

```
head(aggregate(ydat$expression, by=list( ydat$symbol), mean))
```

```
  Group.1           x
1    AAC1  0.52888889
2    AAC3 -0.21628571
3   AAD10  0.43833333
4   AAD14 -0.07166667
5   AAD16  0.24194444
6    AAD4 -0.79166667
```

```
# or
head(aggregate(expression ~ symbol, mean, data=ydat))
```

```
  symbol  expression
1   AAC1  0.52888889
2   AAC3 -0.21628571
3  AAD10  0.43833333
4  AAD14 -0.07166667
5  AAD16  0.24194444
6   AAD4 -0.79166667
```

9.8. Creating a data.frame from scratch

Sometimes it is useful to combine related data into one object. For example, let's simulate some data.

```
smoker = factor(rep(c("smoker", "non-smoker"), each=50))
smoker_numeric = as.numeric(smoker)
x = rnorm(100)
risk = x + 2*smoker_numeric
```

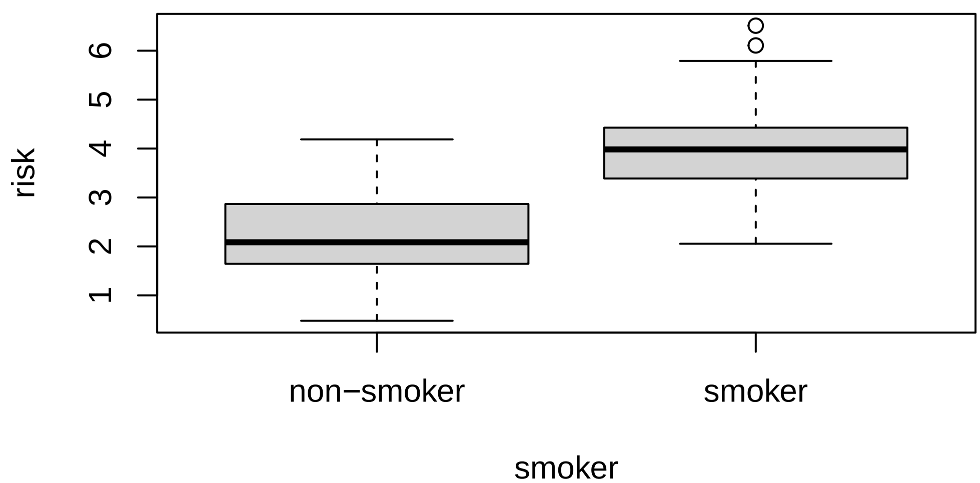
We have two varibles, `risk` and `smoker` that are related. We can make a data.frame out of them:

```
smoker_risk = data.frame(smoker = smoker, risk = risk)
head(smoker_risk)
```

```
  smoker     risk
1 smoker 4.047227
2 smoker 3.710827
3 smoker 3.100671
4 smoker 4.497024
5 smoker 2.723650
6 smoker 2.860481
```

R also has plotting shortcuts that work with data.frames to simplify plotting

```
plot( risk ~ smoker, data=smoker_risk)
```



9.9. Saving a data.frame

Once we have a data.frame of interest, we may want to save it. The most portable way to save a data.frame is to use one of the `write` functions. In this case, let's save the data as a `.csv` file.

```
write.csv(smoker_risk, "smoker_risk.csv")
```

10. Factors

10.1. Factors

A factor is a special type of vector, normally used to hold a categorical variable–such as smoker/nonsmoker, state of residency, zipcode–in many statistical functions. Such vectors have class “factor”. Factors are primarily used in Analysis of Variance (ANOVA) or other situations when “categories” are needed. When a factor is used as a predictor variable, the corresponding indicator variables are created (more later).

Note of caution that factors in R often *appear* to be character vectors when printed, but you will notice that they do not have double quotes around them. They are stored in R as numbers with a key name, so sometimes you will note that the factor *behaves* like a numeric vector.

```
# create the character vector
citizen<-c("uk","us","no","au","uk","us","us","no","au")

# convert to factor
citizenf<-factor(citizen)
citizen
```

```
[1] "uk" "us" "no" "au" "uk" "us" "us" "no" "au"
```

```
citizenf
```

```
[1] uk us no au uk us us no au
Levels: au no uk us
```

```
# convert factor back to character vector
as.character(citizenf)
```

```
[1] "uk" "us" "no" "au" "uk" "us" "us" "no" "au"
```

```
# convert to numeric vector
as.numeric(citizenf)
```

```
[1] 3 4 2 1 3 4 4 2 1
```

R stores many data structures as vectors with "attributes" and "class" (just so you have seen this).

```
attributes(citizenf)
```

```
$levels
[1] "au" "no" "uk" "us"

$class
[1] "factor"
```

```
class(citizenf)
```

```
[1] "factor"
```

```
# note that after unclassing, we can see the
# underlying numeric structure again
unclass(citizenf)
```

```
[1] 3 4 2 1 3 4 4 2 1
attr(,"levels")
[1] "au" "no" "uk" "us"
```

Tabulating factors is a useful way to get a sense of the "sample" set available.

```
table(citizenf)
```

```
citizenf
au no uk us
 2  2  2  3
```

Part III.

Exploratory data analysis

Imagine you're on an adventure, about to embark on a journey into the unknown. You've just been handed a treasure map, with the promise of valuable insights waiting to be discovered. This map is your data set, and the journey is exploratory data analysis (EDA).

As you begin your exploration, you start by getting a feel for the terrain. You take a broad, bird's-eye view of the data, examining its structure and dimensions. Are you dealing with a vast landscape or a small, confined area? Are there any missing pieces in the map that you'll need to account for? Understanding the overall context of your data set is crucial before venturing further.

With a sense of the landscape, you now zoom in to identify key landmarks in the data. You might look for unusual patterns, trends, or relationships between variables. As you spot these landmarks, you start asking questions: What's causing that spike in values? Are these two factors related, or is it just a coincidence? By asking these questions, you're actively engaging with the data and forming hypotheses that could guide future analysis or experiments.

As you continue your journey, you realize that the map alone isn't enough to fully understand the terrain. You need more tools to bring the data to life. You start visualizing the data using charts, plots, and graphs. These visualizations act as your binoculars, allowing you to see patterns and relationships more clearly. Through them, you can uncover the hidden treasures buried within the data.

EDA isn't a linear path from start to finish. As you explore, you'll find yourself circling back to previous points, refining your questions, and digging deeper. The process is iterative, with each new discovery informing the next. And as you go, you'll gain a deeper understanding of the data's underlying structure and potential.

Finally, after your thorough exploration, you'll have a solid foundation to build upon. You'll be better equipped to make informed decisions, test hypotheses, and draw meaningful conclusions. The insights you've gained through EDA will serve as a compass, guiding you towards the true value hidden within your data. And with that, you've successfully completed your journey through exploratory data analysis.

11. Introduction to dplyr: mammal sleep dataset

The dataset we will be using to introduce the *dplyr* package is an updated and expanded version of the mammals sleep dataset. Updated sleep times and weights were taken from V. M. Savage and G. B. West. A quantitative, theoretical framework for understanding mammalian sleep[1].

11.1. Learning goals

- Know that `dplyr` is just a different approach to manipulating data in data.frames.
- List the commonly used `dplyr` verbs and how they can be used to manipulate data.frames.
- Show how to aggregate and summarized data using `dplyr`
- Know what the piping operator, `|>`, is and how it can be used.

11.2. Learning objectives

- Select subsets of the mammal sleep dataset.
- Reorder the dataset.
- Add columns to the dataset based on existing columns.
- Summarize the amount of sleep by categorical variables using `group_by` and `summarize`.

[1]A quantitative, theoretical framework for understanding mammalian sleep. Van M. Savage, Geoffrey B. West. Proceedings of the National Academy of Sciences Jan 2007, 104 (3) 1051-1056; DOI: 10.1073/pnas.0610080104

11.3. What is dplyr?

The *dplyr* package is a specialized package for working with `data.frames` (and the related `tibble`) to transform and summarize tabular data with rows and columns. For another explanation of dplyr see the dplyr package vignette: Introduction to dplyr

11.4. Why Is dplyr userful?

dplyr contains a set of functions–commonly called the dplyr "verbs"–that perform common data manipulations such as filtering for rows, selecting specific columns, re-ordering rows, adding new columns and summarizing data. In addition, dplyr contains a useful function to perform another common task which is the "split-apply-combine" concept.

Compared to base functions in R, the functions in dplyr are often easier to work with, are more consistent in the syntax and are targeted for data analysis around data frames, instead of just vectors.

11.5. Data: Mammals Sleep

The msleep (mammals sleep) data set contains the sleep times and weights for a set of mammals and is available in the dagdata repository on github. This data set contains 83 rows and 11 variables. The data happen to be available as a `data`set in the *ggplot2* package. To get access to the `msleep` dataset, we need to first install the ggplot2 package.

```
install.packages('ggplot2')
```

Then, we can load the library.

```
library(ggplot2)
data(msleep)
```

As with many datasets in R, "help" is available to describe the dataset itself.

```
?msleep
```

The columns are described in the help page, but are included here, also.

column name	Description
name	common name
genus	taxonomic rank
vore	carnivore, omnivore or herbivore?
order	taxonomic rank
conservation	the conservation status of the mammal
sleep_total	total amount of sleep, in hours
sleep_rem	rem sleep, in hours
sleep_cycle	length of sleep cycle, in hours
awake	amount of time spent awake, in hours
brainwt	brain weight in kilograms
bodywt	body weight in kilograms

11.6. dplyr verbs

The dplyr verbs are listed here. There are many other functions available in dplyr, but we will focus on just these.

dplyr verbs	Description
`select()`	select columns
`filter()`	filter rows
`arrange()`	re-order or arrange rows
`mutate()`	create new columns
`summarise()`	summarise values
`group_by()`	allows for group operations in the "split-apply-combine" concept

11.7. Using the dplyr verbs

The two most basic functions are `select()` and `filter()`, which selects columns and filters rows respectively. What are the equivalent ways to select columns without dplyr? And filtering to include only specific rows?

Before proceeding, we need to install the dplyr package:

```
install.packages('dplyr')
```

And then load the library:

```
library(dplyr)
```

```
Attaching package: 'dplyr'

The following objects are masked from 'package:stats':

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

11.7.1. Selecting columns: `select()`

Select a set of columns such as the `name` and the `sleep_total` columns.

```
sleepData <- select(msleep, name, sleep_total)
head(sleepData)
```

```
# A tibble: 6 x 2
  name                       sleep_total
  <chr>                            <dbl>
1 Cheetah                           12.1
2 Owl monkey                        17
3 Mountain beaver                   14.4
4 Greater short-tailed shrew        14.9
5 Cow                                4
6 Three-toed sloth                  14.4
```

To select all the columns *except* a specific column, use the "-" (subtraction) operator (also known as negative indexing). For example, to select all columns except `name`:

```
head(select(msleep, -name))
```

```
# A tibble: 6 x 10
  genus      vore  order    conservation sleep_total sleep_rem sleep_cycle awake
  <chr>      <chr> <chr>    <chr>              <dbl>     <dbl>       <dbl> <dbl>
1 Acinonyx   carni Carnivo~ lc                  12.1      NA        NA      11.9
2 Aotus      omni  Primates <NA>                17         1.8      NA       7
3 Aplodontia herbi Rodentia nt                  14.4       2.4      NA       9.6
4 Blarina    omni  Soricom~ lc                  14.9       2.3       0.133   9.1
5 Bos        herbi Artioda~ domesticated         4         0.7       0.667  20
6 Bradypus   herbi Pilosa   <NA>                14.4       2.2       0.767   9.6
# i 2 more variables: brainwt <dbl>, bodywt <dbl>
```

To select a range of columns by name, use the ":" operator. Note that dplyr allows us to use the column names without quotes and as "indices" of the columns.

```
head(select(msleep, name:order))
```

```
# A tibble: 6 x 4
  name                       genus      vore  order
  <chr>                      <chr>      <chr> <chr>
1 Cheetah                    Acinonyx   carni Carnivora
2 Owl monkey                 Aotus      omni  Primates
3 Mountain beaver            Aplodontia herbi Rodentia
4 Greater short-tailed shrew Blarina    omni  Soricomorpha
5 Cow                        Bos        herbi Artiodactyla
6 Three-toed sloth           Bradypus   herbi Pilosa
```

To select all columns that start with the character string "sl", use the function `starts_with()`.

```
head(select(msleep, starts_with("sl")))
```

```
# A tibble: 6 x 3
  sleep_total sleep_rem sleep_cycle
        <dbl>     <dbl>       <dbl>
1        12.1      NA        NA
2        17         1.8      NA
```

```
3         14.4       2.4      NA
4         14.9       2.3       0.133
5          4         0.7       0.667
6         14.4       2.2       0.767
```

Some additional options to select columns based on a specific criteria include:

1. `ends_with()` = Select columns that end with a character string
2. `contains()` = Select columns that contain a character string
3. `matches()` = Select columns that match a regular expression
4. `one_of()` = Select column names that are from a group of names

11.7.2. Selecting rows: `filter()`

The `filter()` function allows us to filter rows to include only those rows that *match* the filter. For example, we can filter the rows for mammals that sleep a total of more than 16 hours.

```r
filter(msleep, sleep_total >= 16)
```

```
# A tibble: 8 x 11
  name    genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
  <chr>   <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl> <dbl>
1 Owl mo~ Aotus omni  Prim~ <NA>                17         1.8      NA         7
2 Long-n~ Dasy~ carni Cing~ lc                  17.4       3.1       0.383   6.6
3 North ~ Dide~ omni  Dide~ lc                  18         4.9       0.333   6
4 Big br~ Epte~ inse~ Chir~ lc                  19.7       3.9       0.117   4.3
5 Thick-~ Lutr~ carni Dide~ lc                  19.4       6.6      NA         4.6
6 Little~ Myot~ inse~ Chir~ <NA>                19.9       2         0.2     4.1
7 Giant ~ Prio~ inse~ Cing~ en                  18.1       6.1      NA         5.9
8 Arctic~ Sper~ herbi Rode~ lc                  16.6      NA        NA         7.4
# i 2 more variables: brainwt <dbl>, bodywt <dbl>
```

Filter the rows for mammals that sleep a total of more than 16 hours *and* have a body weight of greater than 1 kilogram.

```r
filter(msleep, sleep_total >= 16, bodywt >= 1)
```

```
# A tibble: 3 x 11
  name    genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
  <chr>   <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl> <dbl>
1 Long-n~ Dasy~ carni Cing~ lc                  17.4       3.1       0.383   6.6
2 North ~ Dide~ omni  Dide~ lc                  18         4.9       0.333   6
3 Giant ~ Prio~ inse~ Cing~ en                  18.1       6.1      NA       5.9
# i 2 more variables: brainwt <dbl>, bodywt <dbl>
```

Filter the rows for mammals in the Perissodactyla and Primates taxonomic order. The `%in%` operator is a logical operator that returns `TRUE` for values of a vector that are present *in* a second vector.

```
filter(msleep, order %in% c("Perissodactyla", "Primates"))
```

```
# A tibble: 15 x 11
   name   genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
   <chr>  <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl> <dbl>
 1 Owl m~ Aotus omni  Prim~ <NA>                17         1.8      NA       7
 2 Grivet Cerc~ omni  Prim~ lc                  10         0.7      NA      14
 3 Horse  Equus herbi Peri~ domesticated         2.9       0.6       1      21.1
 4 Donkey Equus herbi Peri~ domesticated         3.1       0.4      NA      20.9
 5 Patas~ Eryt~ omni  Prim~ lc                  10.9       1.1      NA      13.1
 6 Galago Gala~ omni  Prim~ <NA>                 9.8       1.1       0.55   14.2
 7 Human  Homo  omni  Prim~ <NA>                 8         1.9       1.5    16
 8 Mongo~ Lemur herbi Prim~ vu                   9.5       0.9      NA      14.5
 9 Macaq~ Maca~ omni  Prim~ <NA>                10.1       1.2       0.75   13.9
10 Slow ~ Nyct~ carni Prim~ <NA>                11        NA        NA      13
11 Chimp~ Pan   omni  Prim~ <NA>                 9.7       1.4       1.42   14.3
12 Baboon Papio omni  Prim~ <NA>                 9.4       1         0.667  14.6
13 Potto  Pero~ omni  Prim~ lc                  11        NA        NA      13
14 Squir~ Saim~ omni  Prim~ <NA>                 9.6       1.4      NA      14.4
15 Brazi~ Tapi~ herbi Peri~ vu                   4.4       1         0.9    19.6
# i 2 more variables: brainwt <dbl>, bodywt <dbl>
```

You can use the boolean operators (e.g. >, <, >=, <=, !=, %in%) to create the logical tests.

11.8. "Piping" " with |>

It is not unusual to want to perform a set of operations using dplyr. The pipe operator |> allows us to "pipe" the output from one function into the input of the next. While there is nothing special about how R treats operations that are written in a pipe, the idea of piping is to allow us to read multiple functions operating one after another from left-to-right. Without piping, one would either 1) save each step in set of functions as a temporary variable and then pass that variable along the chain or 2) have to "nest" functions, which can be hard to read.

Here's an example we have already used:

```
head(select(msleep, name, sleep_total))
```

```
# A tibble: 6 x 2
  name                       sleep_total
  <chr>                            <dbl>
1 Cheetah                           12.1
2 Owl monkey                        17
3 Mountain beaver                   14.4
4 Greater short-tailed shrew        14.9
5 Cow                                4
6 Three-toed sloth                  14.4
```

Now in this case, we will pipe the msleep data frame to the function that will select two columns (`name` and `sleep\_total`) and then pipe the new data frame to the function `head()`, which will return the head of the new data frame.

```
msleep |>
    select(name, sleep_total) |>
    head()
```

```
# A tibble: 6 x 2
  name                       sleep_total
  <chr>                            <dbl>
1 Cheetah                           12.1
2 Owl monkey                        17
3 Mountain beaver                   14.4
4 Greater short-tailed shrew        14.9
5 Cow                                4
6 Three-toed sloth                  14.4
```

You will soon see how useful the pipe operator is when we start to combine many functions.

Now that you know about the pipe operator (|>), we will use it throughout the rest of this tutorial.

11.8.1. Arrange Or Re-order Rows Using `arrange()`

To arrange (or re-order) rows by a particular column, such as the taxonomic order, list the name of the column you want to arrange the rows by:

```
msleep |> arrange(order) |> head()
```

```
# A tibble: 6 x 11
  name    genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
  <chr>   <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl> <dbl>
1 Tenrec  Tenr~ omni  Afro~ <NA>                15.6       2.3      NA       8.4
2 Cow     Bos   herbi Arti~ domesticated         4         0.7       0.667  20
3 Roe de~ Capr~ herbi Arti~ lc                   3        NA        NA      21
4 Goat    Capri herbi Arti~ lc                   5.3       0.6      NA      18.7
5 Giraffe Gira~ herbi Arti~ cd                   1.9       0.4      NA      22.1
6 Sheep   Ovis  herbi Arti~ domesticated         3.8       0.6      NA      20.2
# i 2 more variables: brainwt <dbl>, bodywt <dbl>
```

Now we will select three columns from msleep, arrange the rows by the taxonomic order and then arrange the rows by sleep_total. Finally, show the head of the final data frame:

```
msleep |>
    select(name, order, sleep_total) |>
    arrange(order, sleep_total) |>
    head()
```

```
# A tibble: 6 x 3
  name     order        sleep_total
  <chr>    <chr>              <dbl>
1 Tenrec   Afrosoricida        15.6
2 Giraffe  Artiodactyla         1.9
3 Roe deer Artiodactyla         3
4 Sheep    Artiodactyla         3.8
```

```
5 Cow      Artiodactyla         4
6 Goat     Artiodactyla         5.3
```

Same as above, except here we filter the rows for mammals that sleep for 16 or more hours, instead of showing the head of the final data frame:

```
msleep |>
    select(name, order, sleep_total) |>
    arrange(order, sleep_total) |>
    filter(sleep_total >= 16)
```

```
# A tibble: 8 x 3
  name                       order           sleep_total
  <chr>                      <chr>                 <dbl>
1 Big brown bat              Chiroptera             19.7
2 Little brown bat           Chiroptera             19.9
3 Long-nosed armadillo       Cingulata              17.4
4 Giant armadillo            Cingulata              18.1
5 North American Opossum     Didelphimorphia        18
6 Thick-tailed opposum       Didelphimorphia        19.4
7 Owl monkey                 Primates               17
8 Arctic ground squirrel     Rodentia               16.6
```

For something slightly more complicated do the same as above, except arrange the rows in the sleep_total column in a descending order. For this, use the function `desc()`

```
msleep |>
    select(name, order, sleep_total) |>
    arrange(order, desc(sleep_total)) |>
    filter(sleep_total >= 16)
```

```
# A tibble: 8 x 3
  name                       order           sleep_total
  <chr>                      <chr>                 <dbl>
1 Little brown bat           Chiroptera             19.9
2 Big brown bat              Chiroptera             19.7
3 Giant armadillo            Cingulata              18.1
4 Long-nosed armadillo       Cingulata              17.4
5 Thick-tailed opposum       Didelphimorphia        19.4
```

```
6 North American Opossum Didelphimorphia        18
7 Owl monkey             Primates               17
8 Arctic ground squirrel Rodentia               16.6
```

11.9. Create New Columns Using `mutate()`

The `mutate()` function will add new columns to the data frame. Create a new column called rem_proportion, which is the ratio of rem sleep to total amount of sleep.

```
msleep |>
    mutate(rem_proportion = sleep_rem / sleep_total) |>
    head()
```

```
# A tibble: 6 x 12
  name    genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
  <chr>   <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl> <dbl>
1 Cheetah Acin~ carni Carn~ lc                  12.1      NA        NA      11.9
2 Owl mo~ Aotus omni  Prim~ <NA>                17         1.8      NA       7
3 Mounta~ Aplo~ herbi Rode~ nt                  14.4       2.4      NA       9.6
4 Greate~ Blar~ omni  Sori~ lc                  14.9       2.3       0.133   9.1
5 Cow     Bos   herbi Arti~ domesticated         4         0.7       0.667  20
6 Three-~ Brad~ herbi Pilo~ <NA>                14.4       2.2       0.767   9.6
# i 3 more variables: brainwt <dbl>, bodywt <dbl>, rem_proportion <dbl>
```

You can add many new columns using mutate (separated by commas). Here we add a second column called bodywt_grams which is the bodywt column in grams.

```
msleep |>
    mutate(rem_proportion = sleep_rem / sleep_total,
           bodywt_grams = bodywt * 1000) |>
    head()
```

```
# A tibble: 6 x 13
  name    genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
  <chr>   <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl> <dbl>
1 Cheetah Acin~ carni Carn~ lc                  12.1      NA        NA      11.9
2 Owl mo~ Aotus omni  Prim~ <NA>                17         1.8      NA       7
3 Mounta~ Aplo~ herbi Rode~ nt                  14.4       2.4      NA       9.6
```

```
4 Greate~ Blar~ omni  Sori~ lc                   14.9       2.3       0.133   9.1
5 Cow     Bos   herbi Arti~ domesticated          4         0.7       0.667  20
6 Three-~ Brad~ herbi Pilo~ <NA>                 14.4       2.2       0.767   9.6
# i 4 more variables: brainwt <dbl>, bodywt <dbl>, rem_proportion <dbl>,
#   bodywt_grams <dbl>
```

Is there a relationship between `rem_proportion` and `bodywt`? How about `sleep_total`?

11.9.1. Create summaries: `summarise()`

The `summarise()` function will create summary statistics for a given column in the data frame such as finding the mean. For example, to compute the average number of hours of sleep, apply the `mean()` function to the column sleep_total and call the summary value avg_sleep.

```
msleep |>
    summarise(avg_sleep = mean(sleep_total))
```

```
# A tibble: 1 x 1
  avg_sleep
      <dbl>
1      10.4
```

There are many other summary statistics you could consider such `sd()`, `min()`, `max()`, `median()`, `sum()`, `n()` (returns the length of vector), `first()` (returns first value in vector), `last()` (returns last value in vector) and `n_distinct()` (number of distinct values in vector).

```
msleep |>
    summarise(avg_sleep = mean(sleep_total),
              min_sleep = min(sleep_total),
              max_sleep = max(sleep_total),
              total = n())
```

```
# A tibble: 1 x 4
  avg_sleep min_sleep max_sleep total
      <dbl>     <dbl>     <dbl> <int>
1      10.4       1.9      19.9    83
```

11.10. Grouping data: `group_by()`

The `group_by()` verb is an important function in dplyr. The `group_by` allows us to use the concept of "split-apply-combine". We literally want to split the data frame by some variable (e.g. taxonomic order), apply a function to the individual data frames and then combine the output. This approach is similar to the `aggregate` function from R, but `group_by` integrates with dplyr.

Let's do that: split the msleep data frame by the taxonomic order, then ask for the same summary statistics as above. We expect a set of summary statistics for each taxonomic order.

```
msleep |>
    group_by(order) |>
    summarise(avg_sleep = mean(sleep_total),
              min_sleep = min(sleep_total),
              max_sleep = max(sleep_total),
              total = n())
```

```
# A tibble: 19 x 5
   order           avg_sleep min_sleep max_sleep total
   <chr>               <dbl>     <dbl>     <dbl> <int>
 1 Afrosoricida        15.6      15.6      15.6      1
 2 Artiodactyla         4.52      1.9       9.1      6
 3 Carnivora           10.1       3.5      15.8     12
 4 Cetacea              4.5       2.7       5.6      3
 5 Chiroptera          19.8      19.7      19.9      2
 6 Cingulata           17.8      17.4      18.1      2
 7 Didelphimorphia     18.7      18        19.4      2
 8 Diprotodontia       12.4      11.1      13.7      2
 9 Erinaceomorpha      10.2      10.1      10.3      2
10 Hyracoidea           5.67      5.3       6.3      3
11 Lagomorpha           8.4       8.4       8.4      1
12 Monotremata          8.6       8.6       8.6      1
13 Perissodactyla       3.47      2.9       4.4      3
14 Pilosa              14.4      14.4      14.4      1
15 Primates            10.5       8        17       12
16 Proboscidea          3.6       3.3       3.9      2
17 Rodentia            12.5       7        16.6     22
18 Scandentia           8.9       8.9       8.9      1
19 Soricomorpha        11.1       8.4      14.9      5
```

12. Case Study: Behavioral Risk Factor Surveillance System

12.1. A Case Study on the Behavioral Risk Factor Surveillance System

The Behavioral Risk Factor Surveillance System (BRFSS) is a large-scale health survey conducted annually by the Centers for Disease Control and Prevention (CDC) in the United States. The BRFSS collects information on various health-related behaviors, chronic health conditions, and the use of preventive services among the adult population (18 years and older) through telephone interviews. The main goal of the BRFSS is to identify and monitor the prevalence of risk factors associated with chronic diseases, inform public health policies, and evaluate the effectiveness of health promotion and disease prevention programs. The data collected through BRFSS is crucial for understanding the health status and needs of the population, and it serves as a valuable resource for researchers, policy makers, and healthcare professionals in making informed decisions and designing targeted interventions.

In this chapter, we will walk through an exploratory data analysis (EDA) of the Behavioral Risk Factor Surveillance System dataset using R. EDA is an important step in the data analysis process, as it helps you to understand your data, identify trends, and detect any anomalies before performing more advanced analyses. We will use various R functions and packages to explore the dataset, with a focus on active learning and hands-on experience.

12.2. Loading the Dataset

First, let's load the dataset into R. We will use the read.csv() function from the base R package to read the data and store it in a data frame called brfss. Make sure the CSV file is in your working directory, or provide the full path to the file.

First, we need to get the data. Either download the data from THIS LINK or have R do it directly from the command-line (preferred):

```
download.file('https://raw.githubusercontent.com/seandavi/ITR/master/BRFSS-subset.csv',
              destfile = 'BRFSS-subset.csv')
```

```
path <- file.choose()    # look for BRFSS-subset.csv
```

```
stopifnot(file.exists(path))
brfss <- read.csv(path)
```

12.3. Inspecting the Data

Once the data is loaded, let's take a look at the first few rows of the dataset using the `head()` function:

```
head(brfss)
```

```
  Age   Weight    Sex Height Year
1  31 48.98798 Female 157.48 1990
2  57 81.64663 Female 157.48 1990
3  43 80.28585   Male 177.80 1990
4  72 70.30682   Male 170.18 1990
5  31 49.89516 Female 154.94 1990
6  58 54.43108 Female 154.94 1990
```

This will display the first six rows of the dataset, allowing you to get a feel for the data structure and variable types.

Next, let's check the dimensions of the dataset using the `dim()` function:

```
dim(brfss)
```

```
[1] 20000     5
```

This will return the number of rows and columns in the dataset, which is important to know for subsequent analyses.

12.4. Summary Statistics

Now that we have a basic understanding of the data structure, let's calculate some summary statistics. The `summary()` function in R provides a quick overview of the main statistics for each variable in the dataset:

```
summary(brfss)
```

```
      Age            Weight            Sex                Height     
 Min.   :18.00   Min.   : 34.93   Length:20000       Min.   :105.0  
 1st Qu.:36.00   1st Qu.: 61.69   Class :character   1st Qu.:162.6  
 Median :51.00   Median : 72.57   Mode  :character   Median :168.0  
 Mean   :50.99   Mean   : 75.42                      Mean   :169.2  
 3rd Qu.:65.00   3rd Qu.: 86.18                      3rd Qu.:177.8  
 Max.   :99.00   Max.   :278.96                      Max.   :218.0  
 NA's   :139     NA's   :649                         NA's   :184    
      Year     
 Min.   :1990  
 1st Qu.:1990  
 Median :2000  
 Mean   :2000  
 3rd Qu.:2010  
 Max.   :2010  
```

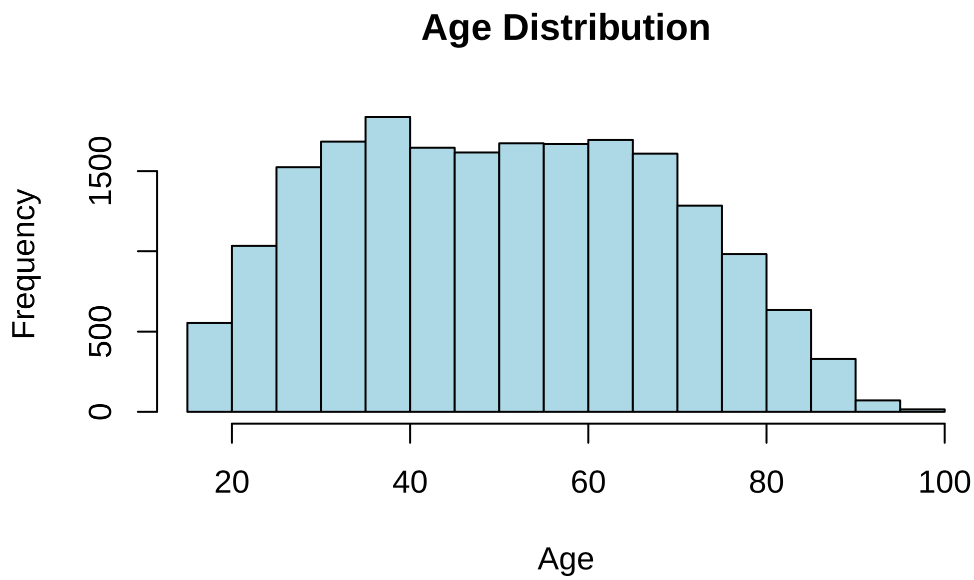
This will display the minimum, first quartile, median, mean, third quartile, and maximum for each numeric variable, and the frequency counts for each factor level for categorical variables.

12.5. Data Visualization

Visualizing the data can help you identify patterns and trends in the dataset. Let's start by creating a histogram of the Age variable using the `hist()` function.

This will create a histogram showing the frequency distribution of ages in the dataset. You can customize the appearance of the histogram by adjusting the parameters within the `hist()` function.

```
hist(brfss$Age, main = "Age Distribution",
     xlab = "Age", col = "lightblue")
```

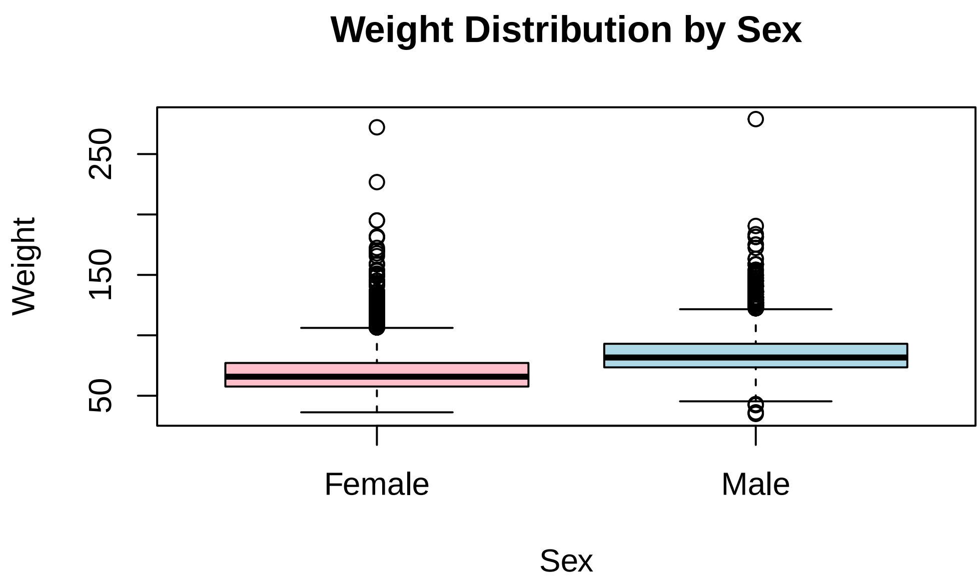


What are the options for a histogram?

The `hist()` function has many options. For example, you can change the number of bins, the color of the bars, the title, and the x-axis label. You can also add a vertical line at the mean or median, or add a normal curve to the histogram. For more information, type `?hist` in the R console.
More generally, it is important to understand the options available for each function you use. You can do this by reading the documentation for the function, which can be accessed by typing `?function_name` or `help("function_name")`in the R console.

Next, let's create a boxplot to compare the distribution of Weight between males and females. We will use the `boxplot()` function for this. This will create a boxplot comparing the weight distribution between males and females. You can customize the appearance of the boxplot by adjusting the parameters within the `boxplot()` function.

```
boxplot(brfss$Weight ~ brfss$Sex, main = "Weight Distribution by Sex",
        xlab = "Sex", ylab = "Weight", col = c("pink", "lightblue"))
```

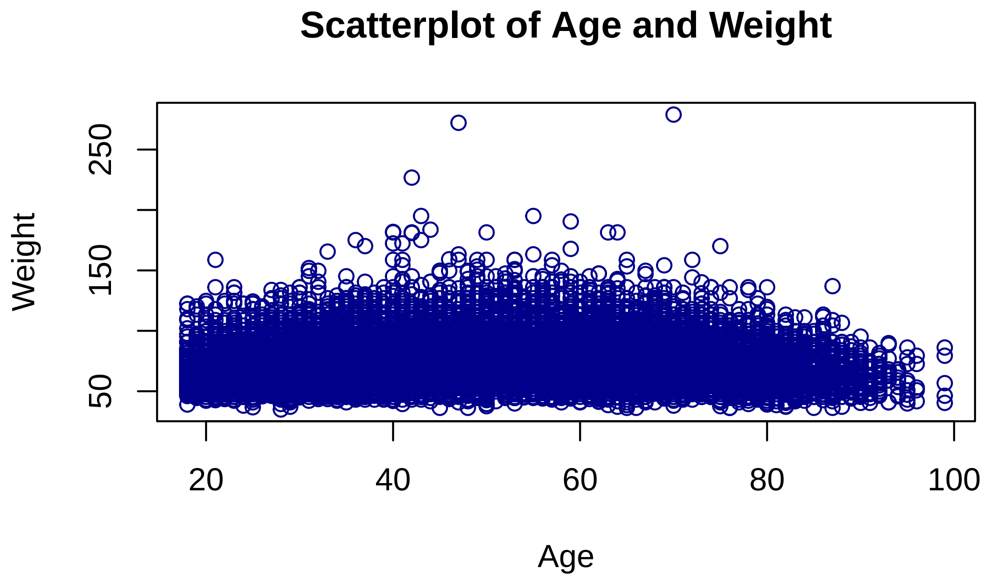


12.6. Analyzing Relationships Between Variables

To further explore the data, let's investigate the relationship between age and weight using a scatterplot. We will use the plot() function for this:

This will create a scatterplot of age and weight, allowing you to visually assess the relationship between these two variables.

```
plot(brfss$Age, brfss$Weight, main = "Scatterplot of Age and Weight",
     xlab = "Age", ylab = "Weight", col = "darkblue")
```



To quantify the strength of the relationship between age and weight, we can calculate the correlation coefficient using the cor() function:

This will return the correlation coefficient between age and weight, which can help you determine whether there is a linear relationship between these variables.

```
cor(brfss$Age, brfss$Weight)
```

```
[1] NA
```

Why does `cor()` give a value of `NA`? What can we do about it? A quick glance at `help("cor")` will give you the answer.

```
cor(brfss$Age, brfss$Weight, use = "complete.obs")
```

```
[1] 0.02699989
```

12.7. Exercises

1. What is the mean weight in this dataset? How about the median? What is the difference between the two? What does this tell you about the distribution of weights in the dataset?

```
mean(brfss$Weight, na.rm = TRUE)
```

```
[1] 75.42455
```

```
median(brfss$Weight, na.rm = TRUE)
```

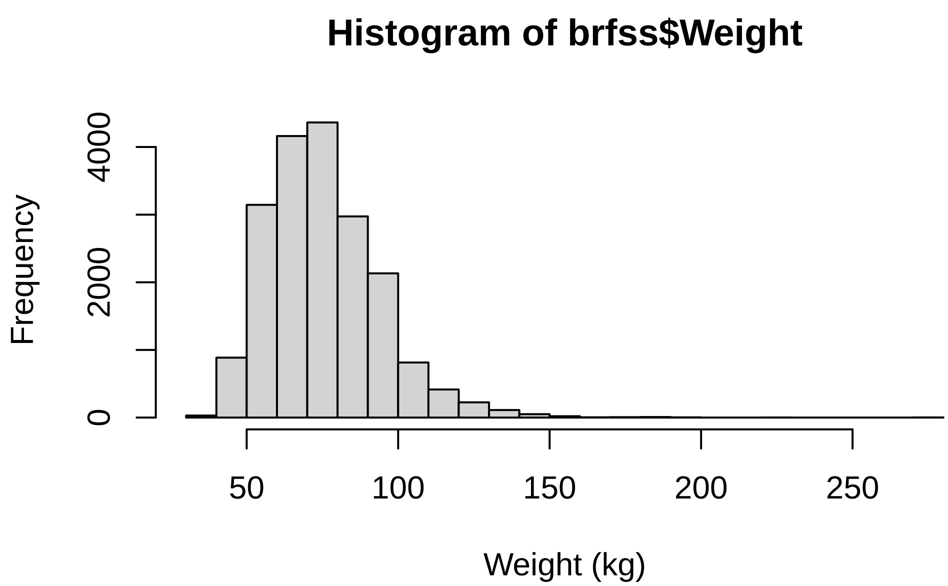
```
[1] 72.57478
```

```
mean(brfss$Weight, na.rm=TRUE) - median(brfss$Weight, na.rm = TRUE)
```

```
[1] 2.849774
```

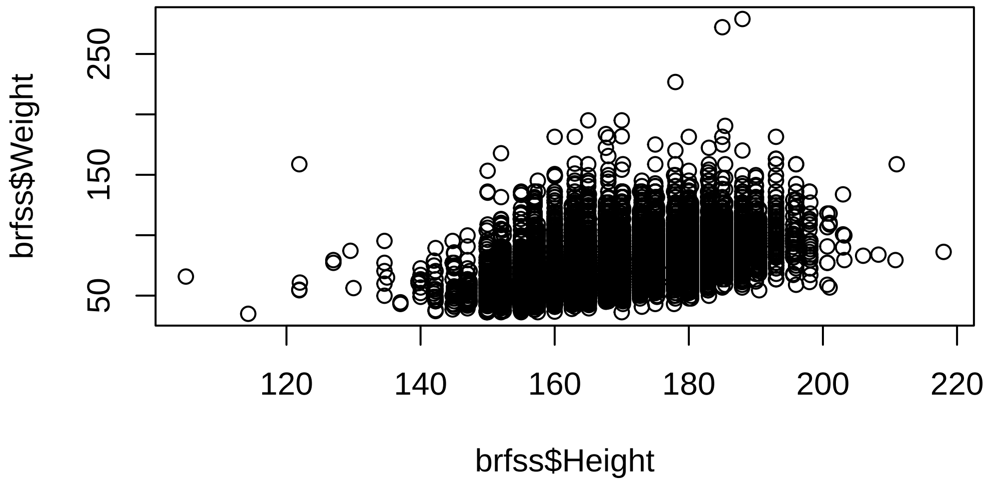
2. Given the findings about the `mean` and `median` in the previous exercise, use the `hist()` function to create a histogram of the weight distribution in this dataset. How would you describe the shape of this distribution?

```
hist(brfss$Weight, xlab="Weight (kg)", breaks = 30)
```



3. Use `plot()` to examine the relationship between height and weight in this dataset.

```
plot(brfss$Height, brfss$Weight)
```



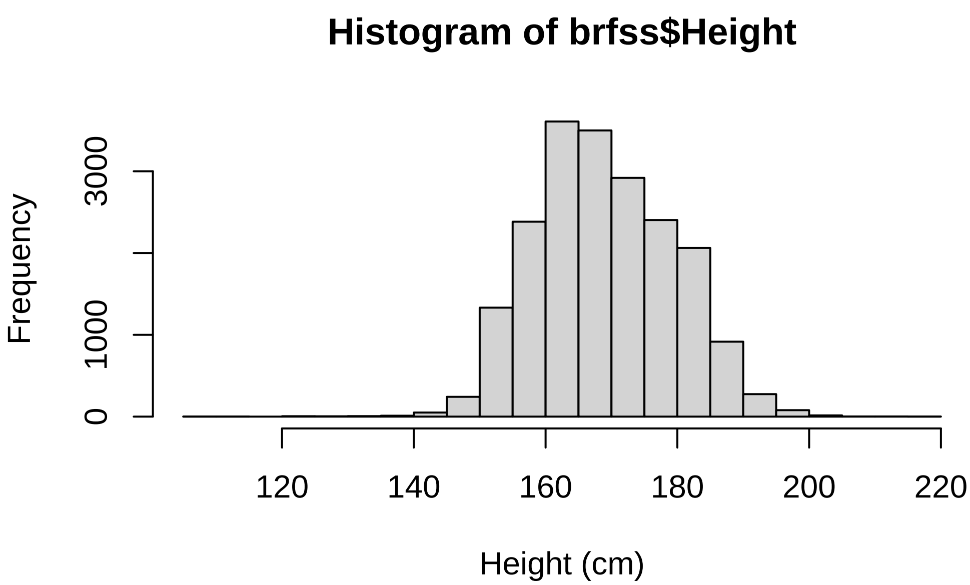
4. What is the correlation between height and weight? What does this tell you about the relationship between these two variables?

```r
cor(brfss$Height, brfss$Weight, use = "complete.obs")
```

```
[1] 0.5140928
```

5. Create a histogram of the height distribution in this dataset. How would you describe the shape of this distribution?

```r
hist(brfss$Height, xlab="Height (cm)", breaks = 30)
```



12.8. Conclusion

In this chapter, we have demonstrated how to perform an exploratory data analysis on the Behavioral Risk Factor Surveillance System dataset using R. We covered data loading, inspection, summary statistics, visualization, and the analysis of relationships between variables. By actively engaging with the R code and data, you have gained valuable experience in using R for EDA and are well-equipped to tackle more complex analyses in your future work.

Remember that EDA is just the beginning of the data analysis process, and further statistical modeling and hypothesis testing will likely be necessary to draw meaningful conclusions from your data. However, EDA is a crucial step in understanding your data and informing your subsequent analyses.

12.9. Learn about the data

Using the data exploration techniques you have seen to explore the brfss dataset.

- summary()
- dim()
- colnames()
- head()
- tail()
- class()
- View()

You may want to investigate individual columns visually using plotting like `hist()`. For categorical data, consider using something like `table()`.

12.10. Clean data

R read `Year` as an integer value, but it's really a `factor`

```
brfss$Year <- factor(brfss$Year)
```

12.11. Weight in 1990 vs. 2010 Females

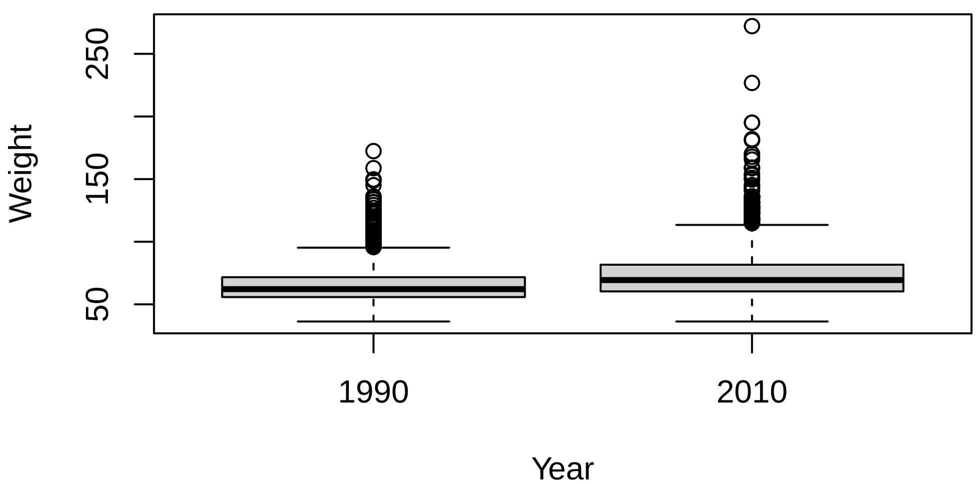
- Create a subset of the data

```
brfssFemale <- brfss[brfss$Sex == "Female",]
summary(brfssFemale)
```

```
      Age            Weight           Sex                Height     
 Min.   :18.00   Min.   : 36.29   Length:12039       Min.   :105.0  
 1st Qu.:37.00   1st Qu.: 57.61   Class :character   1st Qu.:157.5  
 Median :52.00   Median : 65.77   Mode  :character   Median :163.0  
 Mean   :51.92   Mean   : 69.05                      Mean   :163.3  
 3rd Qu.:67.00   3rd Qu.: 77.11                      3rd Qu.:168.0  
 Max.   :99.00   Max.   :272.16                      Max.   :200.7  
 NA's   :103     NA's   :560                         NA's   :140    
   Year     
 1990:5718  
 2010:6321  
```

- Visualize

```
plot(Weight ~ Year, brfssFemale)
```



- Statistical test

```
t.test(Weight ~ Year, brfssFemale)
```

```

	Welch Two Sample t-test

data:  Weight by Year
t = -27.133, df = 11079, p-value < 2.2e-16
alternative hypothesis: true difference in means between group 1990 and group 2010 is not e
95 percent confidence interval:
 -8.723607 -7.548102
sample estimates:
mean in group 1990 mean in group 2010 
          64.81838           72.95424 
```

12.12. Weight and height in 2010 Males

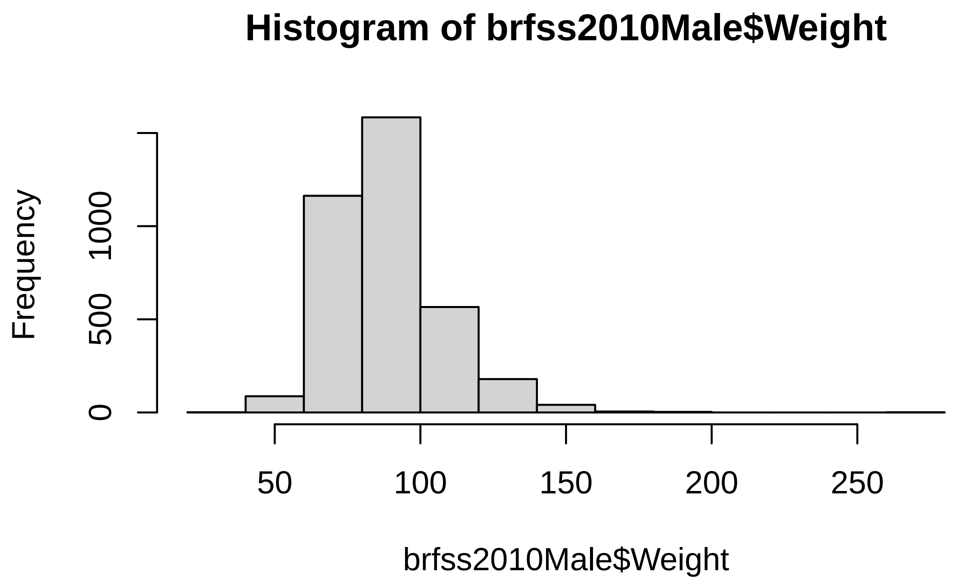
- Create a subset of the data

```
brfss2010Male <- subset(brfss,  Year == 2010 & Sex == "Male")
summary(brfss2010Male)
```

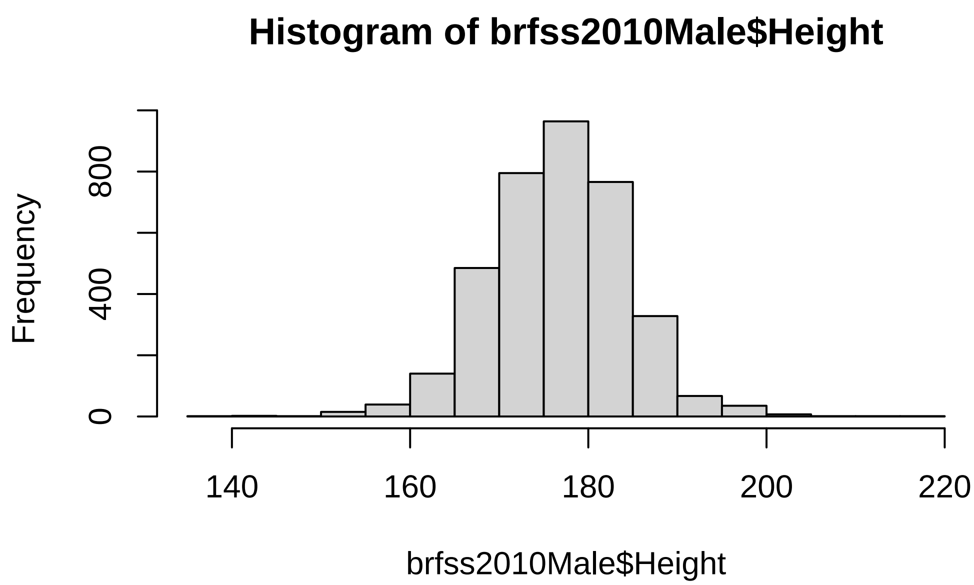
```
      Age            Weight           Sex                Height        Year     
 Min.   :18.00   Min.   : 36.29   Length:3679        Min.   :135   1990:   0  
 1st Qu.:45.00   1st Qu.: 77.11   Class :character   1st Qu.:173   2010:3679  
 Median :57.00   Median : 86.18   Mode  :character   Median :178              
 Mean   :56.25   Mean   : 88.85                      Mean   :178              
 3rd Qu.:68.00   3rd Qu.: 99.79                      3rd Qu.:183              
 Max.   :99.00   Max.   :278.96                      Max.   :218              
 NA's   :30      NA's   :49                          NA's   :31               
```

- Visualize the relationship

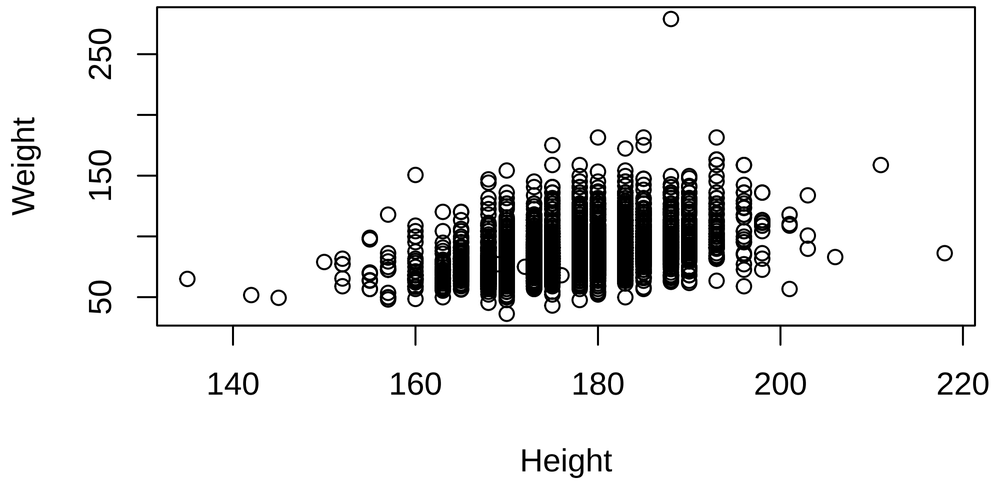
```
hist(brfss2010Male$Weight)
```




```r
hist(brfss2010Male$Height)
```




```r
plot(Weight ~ Height, brfss2010Male)
```



- Fit a linear model (regression)

```r
fit <- lm(Weight ~ Height, brfss2010Male)
fit
```

```

Call:
lm(formula = Weight ~ Height, data = brfss2010Male)

Coefficients:
(Intercept)       Height
   -86.8747       0.9873
```

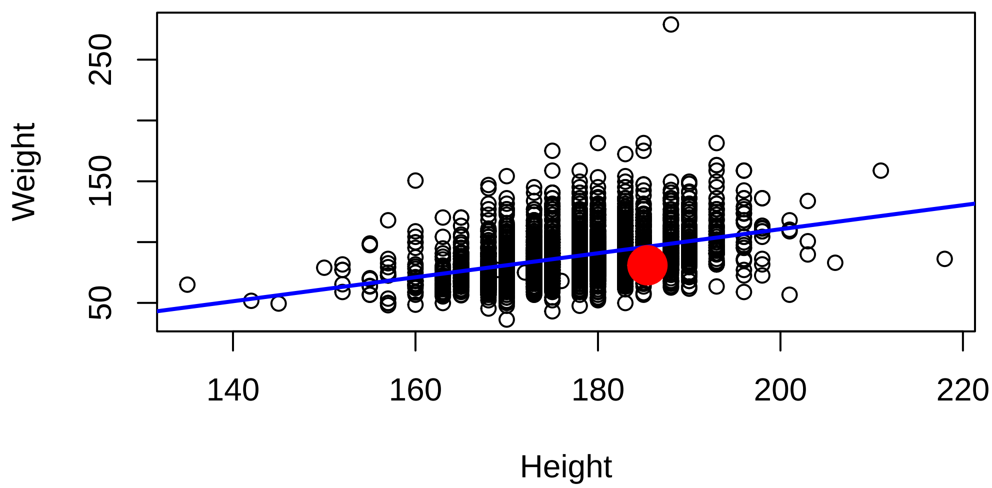
Summarize as ANOVA table

```r
anova(fit)
```

```
Analysis of Variance Table

Response: Weight
            Df  Sum Sq Mean Sq F value    Pr(>F)
Height       1  197664  197664   693.8 < 2.2e-16 ***
Residuals 3617 1030484     285
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

- Plot points, superpose fitted regression line; where am I?

```
plot(Weight ~ Height, brfss2010Male)
abline(fit, col="blue", lwd=2)
# Substitute your own weight and height...
points(73 * 2.54, 178 / 2.2, col="red", cex=4, pch=20)
```



- Class and available 'methods'

```
class(fit)                 # 'noun'
methods(class=class(fit))  # 'verb'
```

- Diagnostics

```
plot(fit)
# Note that the "plot" above does not have a ".lm"
# However, R will use "plot.lm". Why?
?plot.lm
```

Part IV.

statististics

13. Working with distribution functions

Which values do pnorm, dnorm, qnorm, and rnorm return? How do I remember the difference between these?

I find it helpful to have visual representations of distributions as pictures. It is difficult for me to think of distributions, or differences between probability, density, and quantiles without visualizing the shape of the distribution. So I figured it would be helpful to have a visual guide to pnorm, dnorm, qnorm, and rnorm.

Table 13.1.: Table 1.1: Functions for the normal distribution

Function	Input	Output
pnorm	x	$P(X < x)$
dnorm	x	f(x), or the height of the density curve at x
qnorm	q, a quantile from 0 to 1	x such that $P(X < x) = q$
rnorm	n	n random samples from the distribution

13.1. pnorm

This function gives the probability function for a normal distribution. If you do not specify the mean and standard deviation, R defaults to standard normal. Figure 13.1

```r
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
```

The R help file for pnorm provides the template above. The value you input for q is a value on the x-axis, and the returned value is the area under the distribution curve to the left of that point.

```
Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
i Please use `linewidth` instead.
```

This function gives the probability function for a normal distribution. If you do not specify the mean and standard deviation, R defaults to standard normal.

pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE) The R help file for pnorm provides the template above. The value you input for q is a value on the x-axis, and the returned value is the area under the distribution curve to the left of that point.

The option lower.tail = TRUE tells R to use the area to the left of the given point. This is the default, so will remain true even without entering it. In order to compute the area to the right of the given point, you can either switch to lower.tail = FALSE, or simply calculate 1-pnorm() instead. This is demonstrated below.

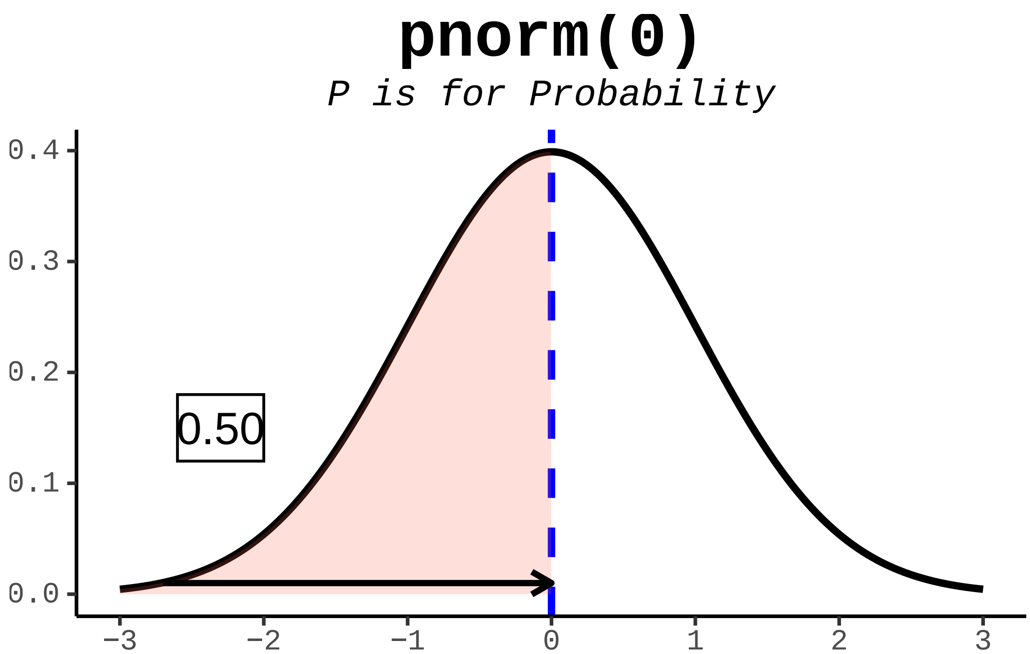


Figure 13.1.: The pnorm function takes a quantile (value on the x-axis) and returns the area under the curve to the left of that value.

The option `lower.tail = TRUE` tells R to use the area to the left of the given point. This is the default, so will remain true even without entering it. In order to compute the area to the right of the given point, you can either switch to `lower.tail = FALSE`, or simply calculate `1-pnorm()` instead.

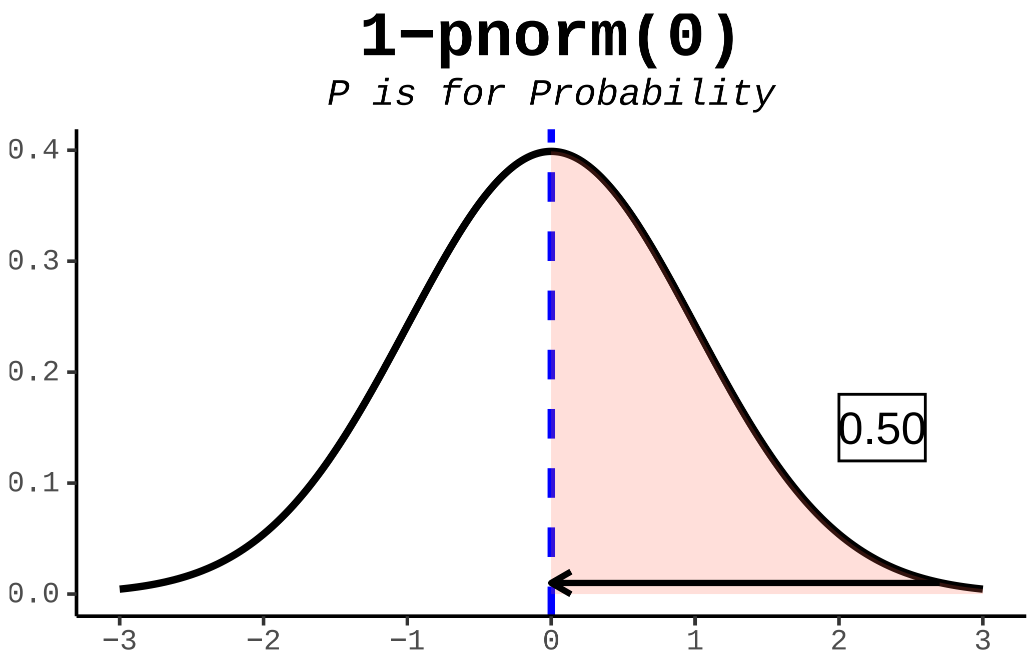


Figure 13.2.: The pnorm function takes a quantile (value on the x-axis) and returns the area under the curve to the left of that value.

13.2. dnorm

This function calculates the probability density function (PDF) for the normal distribution. It gives the probability density (height of the curve) at a specified value (x).

13.3. qnorm

This function calculates the quantiles of the normal distribution. It returns the value (x) corresponding to a specified probability (p). It is the inverse of the`pnorm` function.

13.4. rnorm

```
print(r1)
```

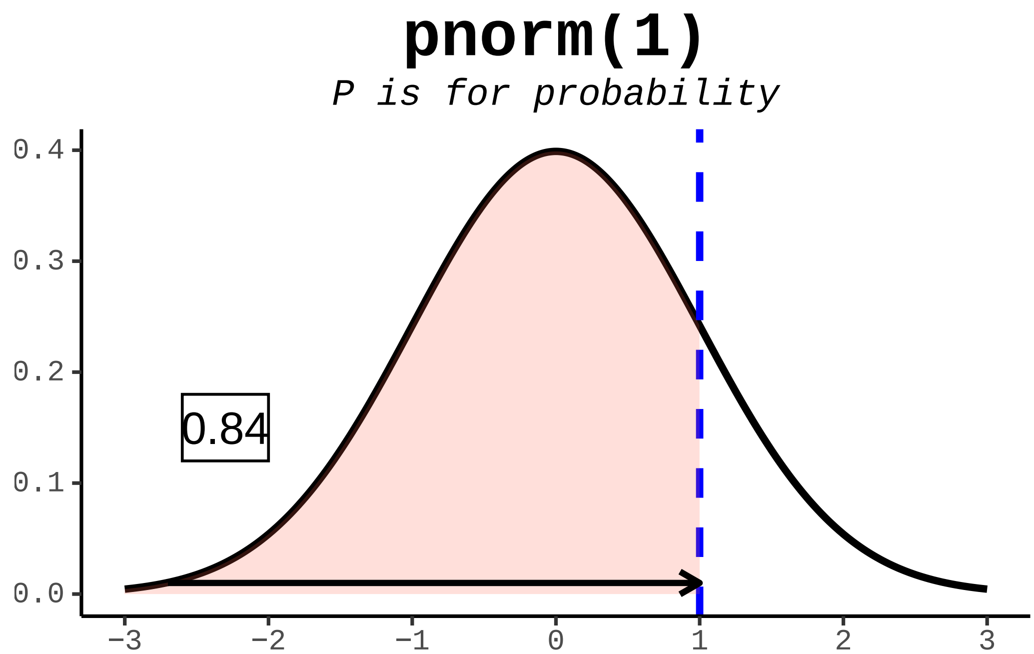


Figure 13.3.: The pnorm function takes a quantile (value on the x-axis) and returns the area under the curve to the left of that value.

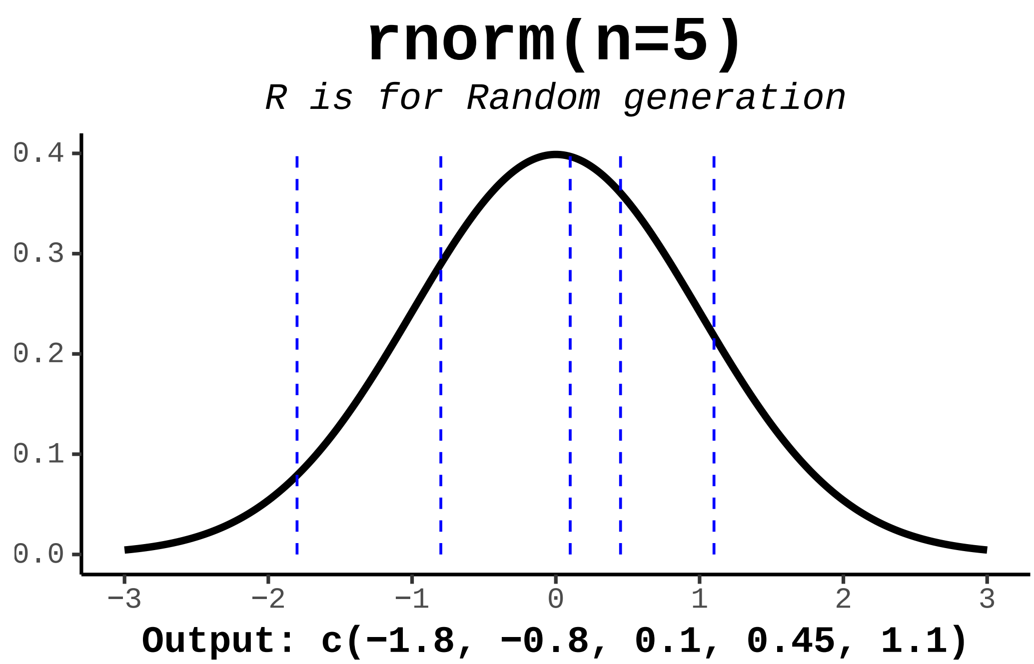


Figure 13.13.: The rnorm function takes a number of samples and returns a vector of random numbers from the normal distribution (with mean=0, sd=1 as defaults)

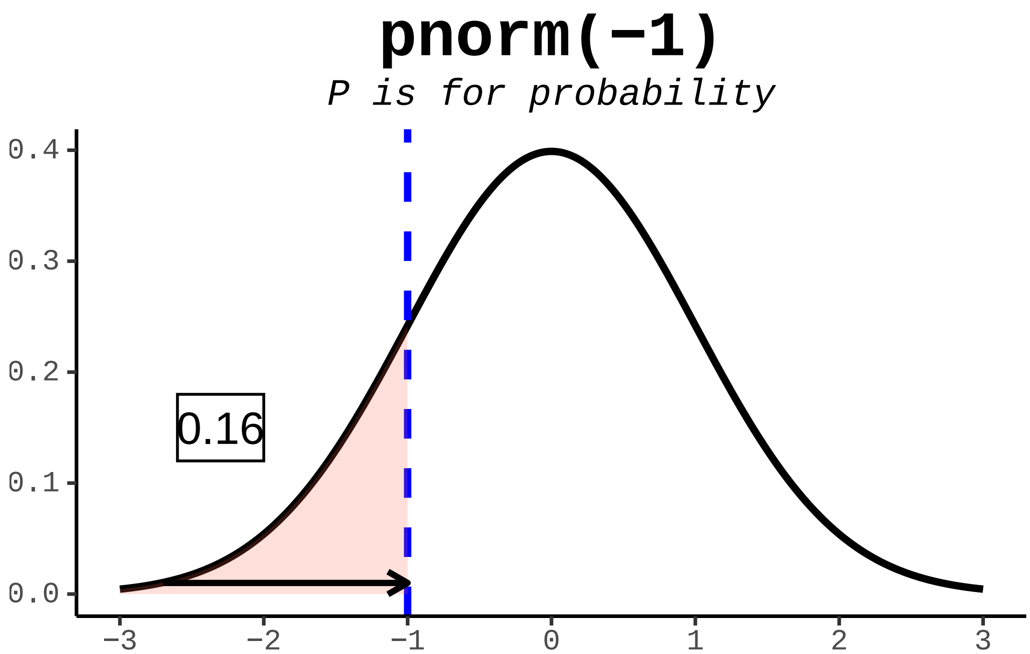


Figure 13.4.: The pnorm function takes a quantile (value on the x-axis) and returns the area under the curve to the left of that value.

13.5. IQ scores

Normal Distribution and its Application with IQ

The normal distribution, also known as the Gaussian distribution, is a continuous probability distribution characterized by its bell-shaped curve. It is defined by two parameters: the mean (μ) and the standard deviation ( ). The mean represents the central tendency of the distribution, while the standard deviation represents the dispersion or spread of the data.

The IQ scores are an excellent example of the normal distribution, as they are designed to follow this distribution pattern. The mean IQ score is set at 100, and the standard deviation is set at 15. This means that the majority of the population (about 68%) have an IQ score between 85 and 115, while 95% of the population have an IQ score between 70 and 130.

- What is the probability of having an IQ score between 85 and 115?

  ```
  pnorm(115, mean = 100, sd = 15) - pnorm(85, mean = 100, sd = 15)
  ```

- What is the 90th percentile of the IQ scores?

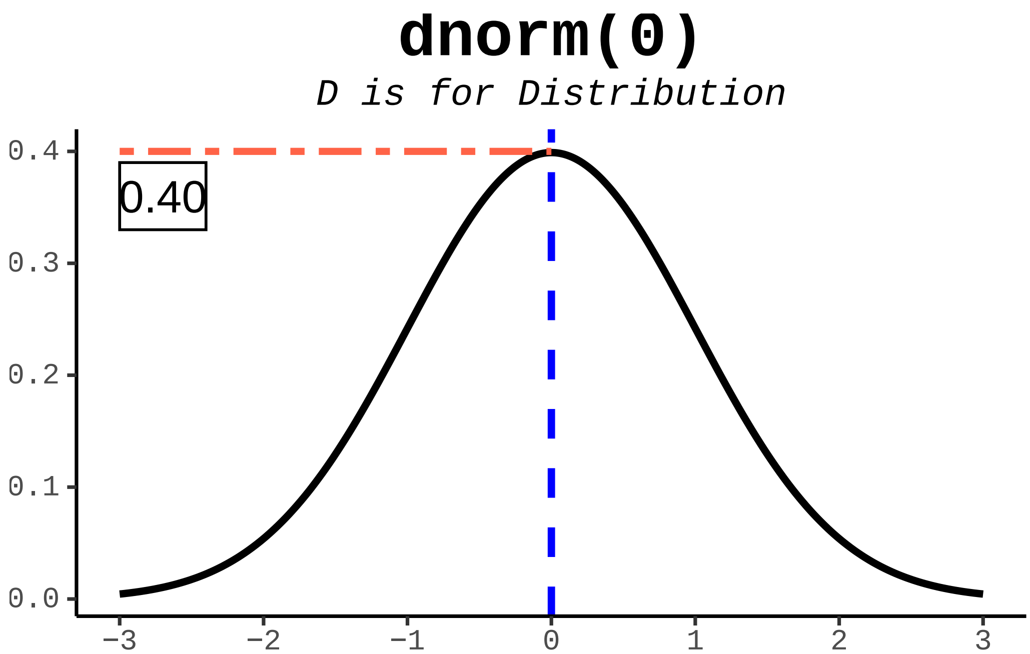


Figure 13.5.: The `dnorm` function returns the height of the normal distribution at a given point.

```
qnorm(0.9, mean = 100, sd = 15)
```

- What is the probability of having an IQ score above 130?

```
1 - pnorm(130, mean = 100, sd = 15)
```

- What is the probability of having an IQ score below 70?

```
pnorm(70, mean = 100, sd = 15)
```

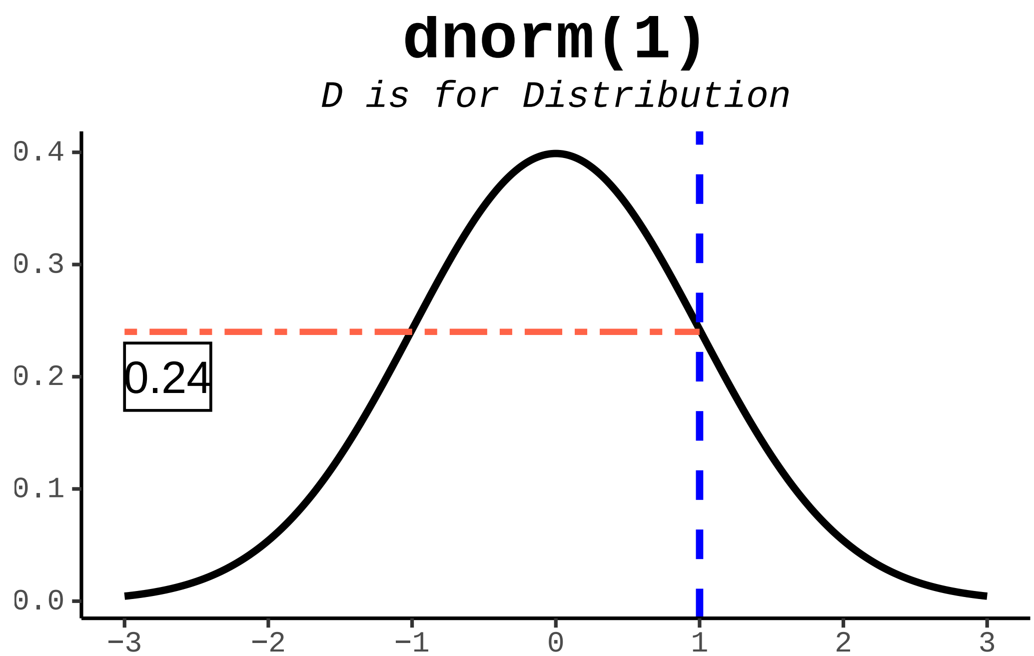


Figure 13.6.: The `dnorm` function returns the height of the normal distribution at a given point.

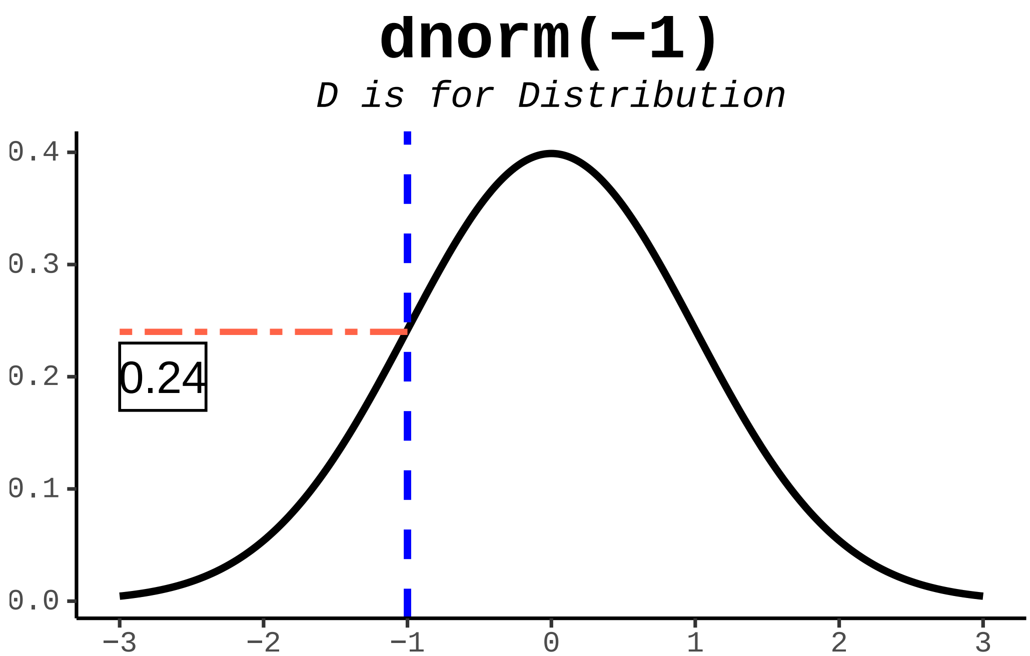


Figure 13.7.: The `dnorm` function returns the height of the normal distribution at a given point.

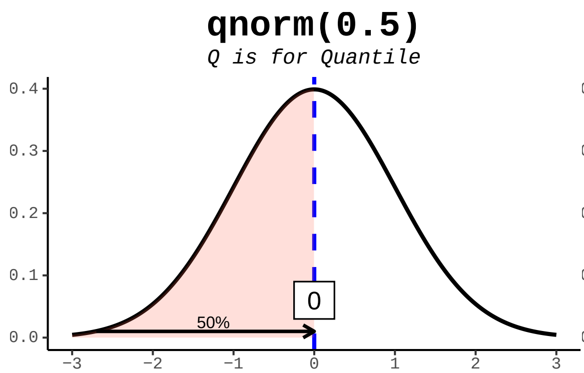


Figure 13.8.: The qnorm function is the inverse of the pnorm function in that it takes a probability and gives the quantile.

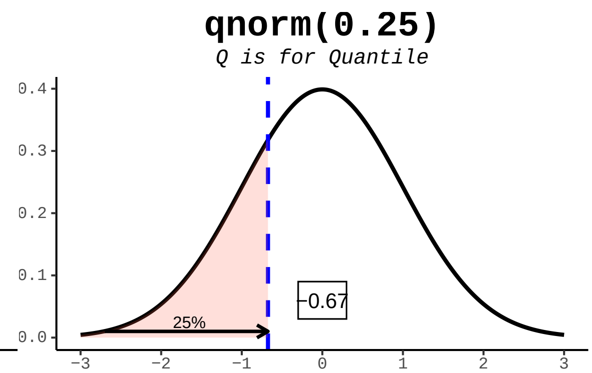


Figure 13.9.: The qnorm function is the inverse of the pnorm function in that it takes a probability and gives the quantile.

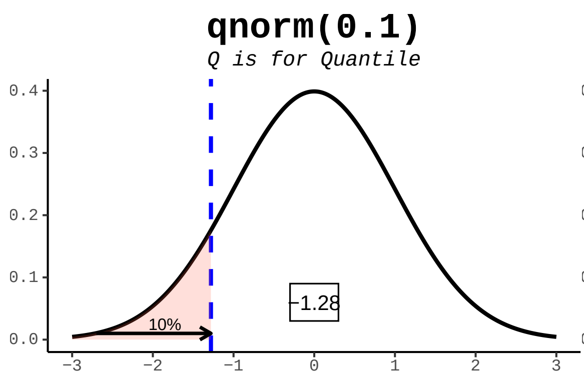


Figure 13.10.: The qnorm function is the inverse of the pnorm function in that it takes a probability and gives the quantile.

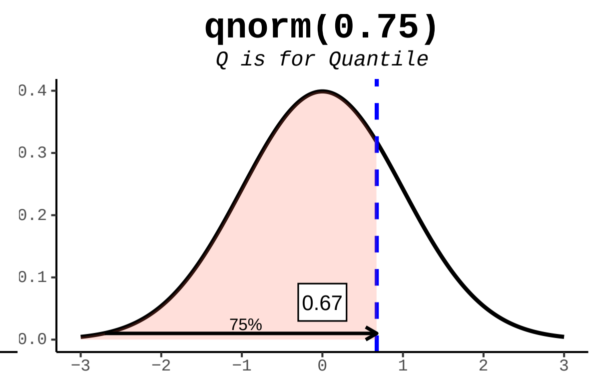


Figure 13.11.: The qnorm function is the inverse of the pnorm function in that it takes a probability and gives the quantile.

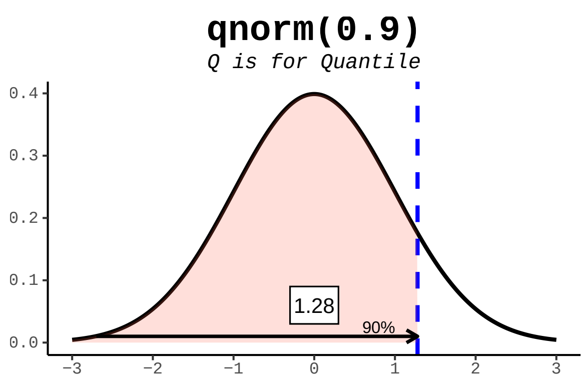


Figure 13.12.: The qnorm function is the inverse of the pnorm function in that it takes a probability and gives the quantile.

14. The t-statistic and t-distribution

14.1. Background

The t-test is a statistical hypothesis test that is commonly used when the data are normally distributed (follow a normal distribution) if the value of the population standard deviation were known. When the population standard deviation is not known and is replaced by an estimate based no the data, the test statistic follows a Student's t distribution.

T-tests are handy hypothesis tests in statistics when you want to compare means. You can compare a sample mean to a hypothesized or target value using a one-sample t-test. You can compare the means of two groups with a two-sample t-test. If you have two groups with paired observations (e.g., before and after measurements), use the paired t-test.

A t-test looks at the t-statistic, the t-distribution values, and the degrees of freedom to determine the statistical significance. To conduct a test with three or more means, we would use an analysis of variance.

The distriubution that the t-statistic follows was described in a famous paper (Student 1908) by "Student", a pseudonym for William Sealy Gosset.

14.2. The Z-score and probability

Before talking about the t-distribution and t-scores, lets review the Z-score, its relation to the normal distribution, and probability.

The Z-score is defined as:

$$Z = \frac{x - \mu}{\sigma} \tag{14.1}$$

where μ is a the population mean from which x is drawn and σ is the population standard deviation (taken as known, not estimated from the data).

The probability of observing a Z score of z or greater can be calculated by $pnorm(z, \mu, \sigma)$.

For example, let's assume that our "population" is known and it truly has a mean 0 and standard deviation 1. If we have observations drawn from that population, we can assign a probability of seeing that observation by random chance *under the assumption that the null hypothesis is **TRUE***.

```
zscore = seq(-5,5,1)
```

For each value of zscore, let's calculate the p-value and put the results in a `data.frame`.

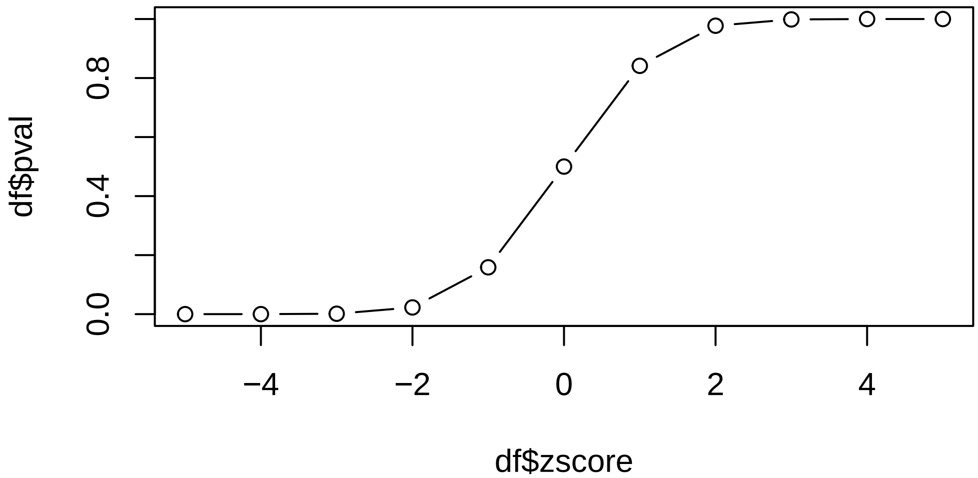
```
df = data.frame(
    zscore = zscore,
    pval   = pnorm(zscore, 0, 1)
)
df
```

```
   zscore         pval
1      -5 2.866516e-07
2      -4 3.167124e-05
3      -3 1.349898e-03
4      -2 2.275013e-02
5      -1 1.586553e-01
6       0 5.000000e-01
7       1 8.413447e-01
8       2 9.772499e-01
9       3 9.986501e-01
10      4 9.999683e-01
11      5 9.999997e-01
```

Why is the p-value of something 5 population standard deviations away from the mean (zscore=5) nearly 1 in this calculation? What is the default for `pnorm` with respect to being one-sided or two-sided?

Let's plot the values of probability vs z-score:

```
plot(df$zscore, df$pval, type='b')
```



This plot is the *empirical* cumulative density function (cdf) for our data. How can we use it? If we know the z-score, we can look up the probability of observing that value. Since we have constructed our experiment to follow the standard normal distribution, this cdf also represents the cdf of the standard normal distribution.

14.2.1. Small diversion: two-sided pnorm function

The `pnorm` function returns the "one-sided" probability of having a value at least as extreme as the observed x and uses the "lower" tail by default. Let's create a function that computes two-sided p-values.

1. Take the absolute value of x
2. Compute `pnorm` with `lower.tail=FALSE` so we get lower p-values with larger values of x.
3. Since we want to include both tails, we need to multiply the area (probability) returned by pnorm by 2.

```
twosidedpnorm = function(x,mu=0,sd=1) {
    2*pnorm(abs(x),mu,sd,lower.tail=FALSE)
}
```

And we can test this to see how likely it is to be 2 or 3 standard deviations from the mean:

```
twosidedpnorm(2)
```

```
[1] 0.04550026
```

```r
twosidedpnorm(3)
```

```
[1] 0.002699796
```

14.3. The t-distribution

We spent time above working with z-scores and probability. An important aspect of working with the normal distribution is that we MUST assume that we know the standard deviation. Remember that the Z-score is defined as:

$$Z = \frac{x - \mu}{\sigma}$$

The formula for the *population* standard deviation is:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(xi - \mu)^2} \tag{14.2}$$

In general, the population standard deviation is taken as "known" as we did above.

If we do not but only have a *sample* from the population, instead of using the Z-score, we use the t-score defined as:

$$t = \frac{x - \bar{x}}{s} \tag{14.3}$$

This looks quite similar to the formula for Z-score, but here we have to *estimate* the standard deviation, s from the data. The formula for s is:

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2} \tag{14.4}$$

Since we are estimating the standard deviation from the data, this leads to extra variability that shows up as "fatter tails" for smaller sample sizes than for larger sample sizes. We can see this by comparing the *t-distribution* for various numbers of degrees of freedom (sample sizes).

We can look at the effect of sample size on the distributions graphically by looking at the densities for 3, 5, 10, 20 degrees of freedom and the normal distribution:

```
library(dplyr)
library(ggplot2)
t_values = seq(-6,6,0.01)
df = data.frame(
    value = t_values,
    t_3   = dt(t_values,3),
    t_6   = dt(t_values,6),
    t_10  = dt(t_values,10),
    t_20  = dt(t_values,20),
    Normal= dnorm(t_values)
) |>
    tidyr::gather("Distribution", "density", -value)
ggplot(df, aes(x=value, y=density, color=Distribution)) +
    geom_line()
```

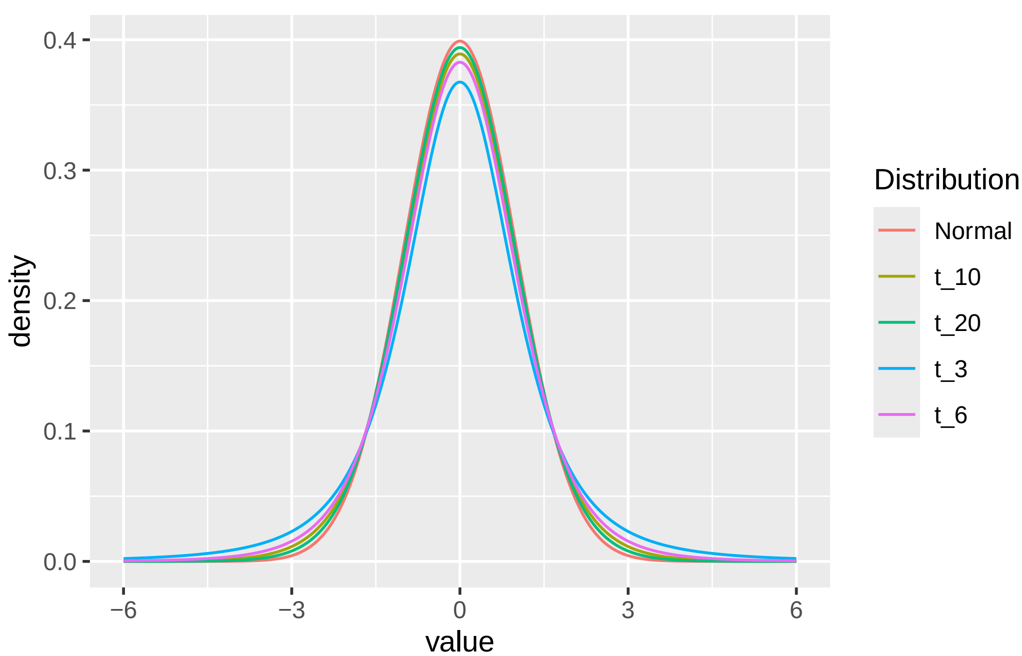
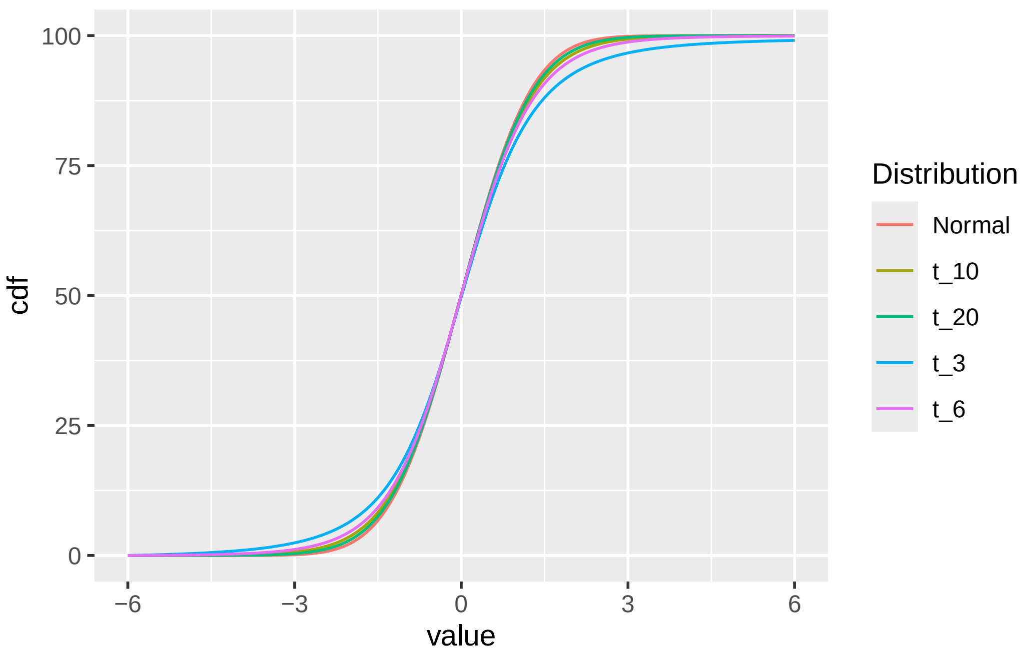


Figure 14.1.: t-distributions for various degrees of freedom. Note that the tails are fatter for smaller degrees of freedom, which is a result of estimating the standard deviation from the data.

The `dt` and `dnorm` functions give the density of the distributions for each point.

```
df2 = df |>
    group_by(Distribution) |>
    arrange(value) |>
    mutate(cdf=cumsum(density))
ggplot(df2, aes(x=value, y=cdf, color=Distribution)) +
    geom_line()
```



14.3.1. p-values based on Z vs t

When we have a "sample" of data and want to compute the statistical significance of the difference of the mean from the population mean, we calculate the standard deviation of the sample means (standard error).

$$z = \frac{x - \mu}{\sigma/\sqrt{n}}$$

Let's look at the relationship between the p-values of Z (from the normal distribution) vs t for a **sample** of data.

```
set.seed(5432)
samp = rnorm(5,mean = 0.5)
z = sqrt(length(samp)) * mean(samp) #simplifying assumption (sigma=1, mu=0)
```

And the p-value if we assume we know the standard deviation:

```
pnorm(z, lower.tail = FALSE)
```

```
[1] 0.02428316
```

In reality, we don't know the standard deviation, so we have to estimate it from the data. We can do this by calculating the sample standard deviation:

```
ts = sqrt(length(samp)) * mean(samp) / sd(samp)
pnorm(ts, lower.tail = FALSE)
```

```
[1] 0.0167297
```

```
pt(ts,df = length(samp)-1, lower.tail = FALSE)
```

```
[1] 0.0503001
```

14.3.2. Experiment

When sampling from a normal distribution, we often calculate p-values to test hypotheses or determine the statistical significance of our results. The p-value represents the probability of obtaining a test statistic as extreme or more extreme than the one observed, under the null hypothesis.

In a typical scenario, we assume that the population mean and standard deviation are known. However, in many real-life situations, we don't know the true population standard deviation, and we have to estimate it using the sample standard deviation (Equation 14.4). This estimation introduces some uncertainty into our calculations, which affects the p-values. When we include an estimate of the standard deviation, we switch from using the standard normal (z) distribution to the t-distribution for calculating p-values.

What would happen if we used the normal distribution to calculate p-values when we use the sample standard deviation? Let's find out!

1. Simulate a bunch of samples of size `n` from the standard normal distribution
2. Calculate the p-value distribution for those samples based on the normal.
3. Calculate the p-value distribution for those samples based on the normal, but with the *estimated* standard deviation.
4. Calculate the p-value distribution for those samples based on the t-distribution.

Create a function that draws a sample of size `n` from the standard normal distribution.

```
zf = function(n) {
    samp = rnorm(n)
    z = sqrt(length(samp)) * mean(samp) / 1 #simplifying assumption (sigma=1, mu=0)
    z
}
```

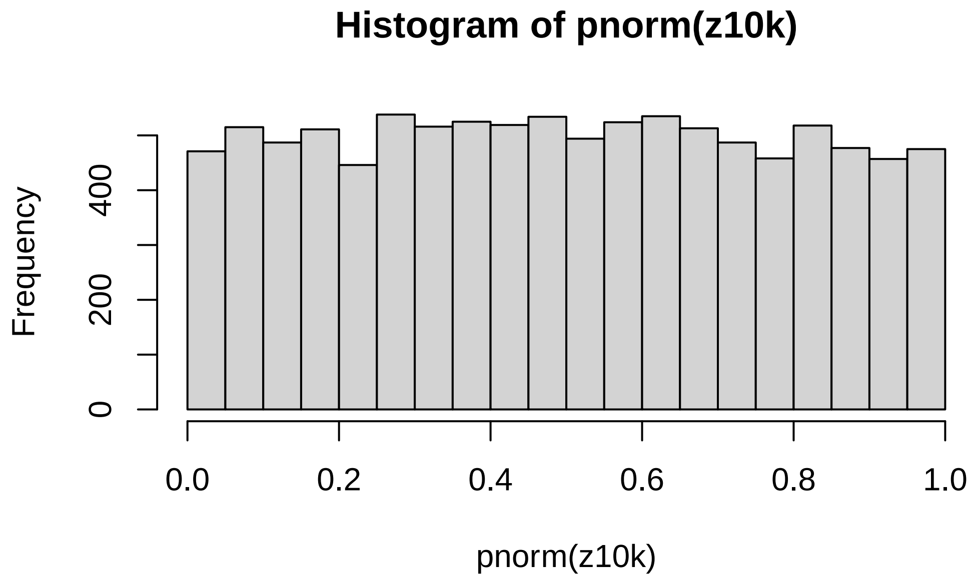
And give it a try:

```
zf(5)
```

```
[1] 0.7406094
```

Perform 10000 replicates of our sampling and z-scoring. We are using the assumption that we know the population standard deviation; in this case, we do know since we are sampling from the standard normal distribution.

```
z10k = replicate(10000,zf(5))
hist(pnorm(z10k))
```

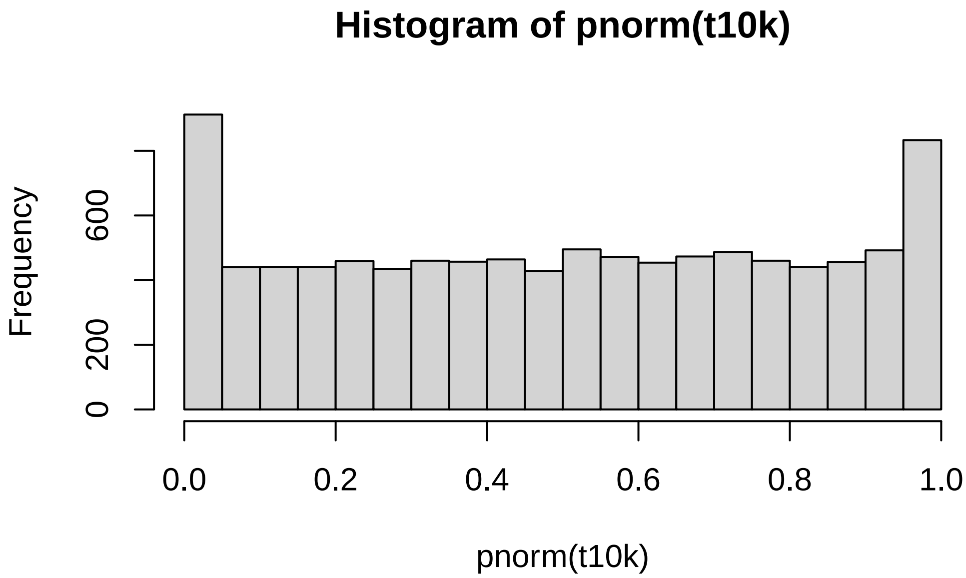


And do the same, but now creating a t-score function. We are using the assumption that we *don't* know the population standard deviation; in this case, we must estimate it from the data. Note the difference in the calculation of the t-score (`ts`) as compared to the z-score (`z`).

```
tf = function(n) {
    samp = rnorm(n)
    # now, using the sample standard deviation since we
    # "don't know" the population standard deviation
    ts = sqrt(length(samp)) * mean(samp) / sd(samp)
    ts
}
```

If we use those t-scores and calculate the p-values based on the normal distribution, the histogram of those p-values looks like:

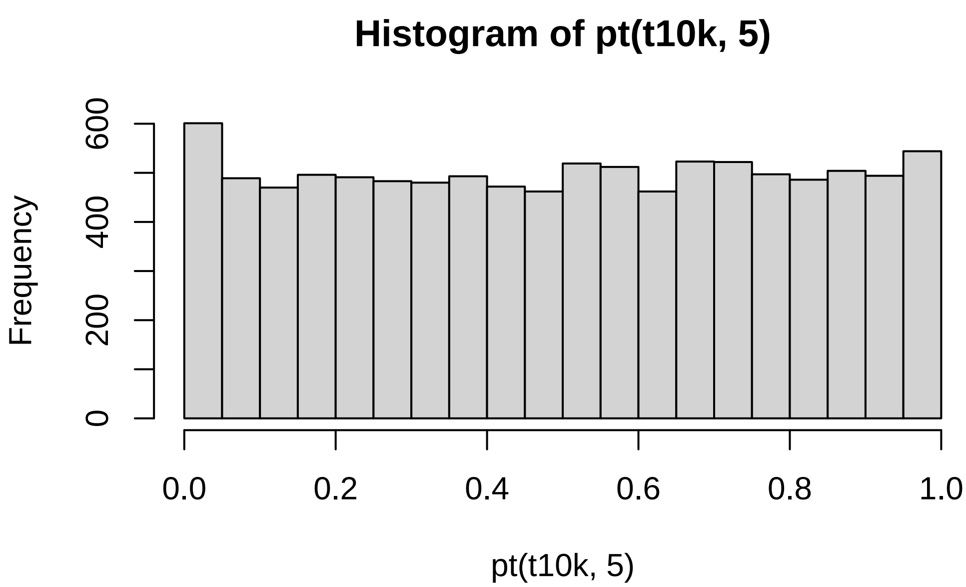
```
t10k = replicate(10000,tf(5))
hist(pnorm(t10k))
```



Since we are using the normal distribution to calculate the p-values, we are, in effect, assuming that we know the population standard deviation. This assumption is incorrect, and we can see that the p-values are not uniformly distributed between 0 and 1.

If we use those t-scores and calculate the p-values based on the t-distribution, the histogram of those p-values looks like:

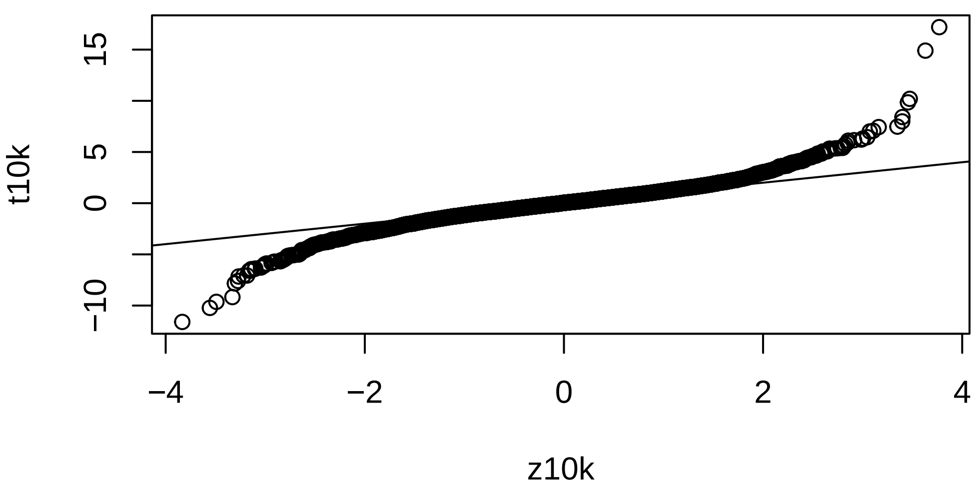
```
hist(pt(t10k,5))
```



Now, the p-values are uniformly distributed between 0 and 1, as expected.

What is a qqplot and how do we use it? A qqplot is a plot of the quantiles of two distributions against each other. If the two distributions are identical, the points will fall on a straight line. If the two distributions are different, the points will deviate from the straight line. We can use a qqplot to compare the t-distribution to the normal distribution. If the t-distribution is identical to the normal distribution, the points will fall on a straight line. If the t-distribution is different from the normal distribution, the points will deviate from the straight line. In this case, we can see that the t-distribution is different from the normal distribution, as the points deviate from the straight line. What would happen if we increased the sample size? The t-distribution would approach the normal distribution, and the points would fall closer and closer to the straight line.

```
qqplot(z10k,t10k)
abline(0,1)
```



14.4. Summary of t-distribution vs normal distribution

The t-distribution is a family of probability distributions that depends on a parameter called degrees of freedom, which is related to the sample size. The t-distribution approaches the standard normal distribution as the sample size increases but has heavier tails for smaller sample sizes. This means that the t-distribution is more conservative in calculating p-values for small samples, making it harder to reject the null hypothesis. Including an estimate of the standard deviation changes the way we calculate p-values by switching from the standard normal distribution to the t-distribution, which accounts for the uncertainty introduced by estimating the population standard deviation from the sample. This adjustment is particularly important for small sample sizes, as it provides a more accurate assessment of the statistical significance of our results.

14.5. t.test

14.5.1. One-sample

We are going to use the `t.test` function to perform a one-sample t-test. The `t.test` function takes a vector of values as input that represents the sample values. In this case, we'll simulate our sample using the `rnorm` function and presume that our "effect-size" is 1.

```
x = rnorm(20,1)
# small sample
# Just use the first 5 values of the sample
t.test(x[1:5])
```

```

    One Sample t-test

data:  x[1:5]
t = 0.97599, df = 4, p-value = 0.3843
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 -1.029600  2.145843
sample estimates:
mean of x
0.5581214
```

In this case, we set up the experiment so that the null hypothesis is true (the true mean is not zero, but actually 1). However, we only have a small sample size that leads to a modest p-value.

Increasing the sample size allows us to see the effect more clearly.

```
t.test(x[1:20])
```

```
	One Sample t-test

data:  x[1:20]
t = 3.8245, df = 19, p-value = 0.001144
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 0.3541055 1.2101894
sample estimates:
mean of x
0.7821474
```

14.5.2. two-sample

```
x = rnorm(10,0.5)
y = rnorm(10,-0.5)
t.test(x,y)
```

```
	Welch Two Sample t-test

data:  x and y
t = 3.4296, df = 17.926, p-value = 0.003003
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.5811367 2.4204048
sample estimates:
 mean of x  mean of y
 0.7039205 -0.7968502
```

14.5.3. from a data.frame

In some situations, you may have data and groups as columns in a data.frame. See the following data.frame, for example

```
df = data.frame(value=c(x,y),group=as.factor(rep(c('g1','g2'),each=10)))
df
```

```
         value group
1   1.12896674    g1
2  -1.26838101    g1
3   1.04577597    g1
4   1.69075585    g1
5   0.18672204    g1
6   1.99715092    g1
7   1.15424947    g1
8   0.37671442    g1
9  -0.09565723    g1
10  0.82290783    g1
11 -1.48530261    g2
12 -1.29200440    g2
13 -0.18778362    g2
14  0.59205742    g2
15 -2.10065248    g2
16 -0.29961560    g2
17 -0.38985115    g2
18 -2.47126235    g2
19 -0.63654380    g2
20  0.30245611    g2
```

R allows us to perform a t-test using the `formula` notation.

```r
t.test(value ~ group, data=df)
```

```

	Welch Two Sample t-test

data:  value by group
t = 3.4296, df = 17.926, p-value = 0.003003
alternative hypothesis: true difference in means between group g1 and group g2 is not equal
95 percent confidence interval:
 0.5811367 2.4204048
sample estimates:
mean in group g1 mean in group g2 
       0.7039205       -0.7968502 
```

You read that as `value` **is a function of** `group`. In practice, this will do a t-test between the `values` in `g1` vs `g2`.

14.5.4. Equivalence to linear model

```
t.test(value ~ group, data=df, var.equal=TRUE)
```

```

	Two Sample t-test

data:  value by group
t = 3.4296, df = 18, p-value = 0.002989
alternative hypothesis: true difference in means between group g1 and group g2 is not equal
95 percent confidence interval:
 0.5814078 2.4201337
sample estimates:
mean in group g1 mean in group g2 
       0.7039205       -0.7968502 
```

This is *equivalent* to:

```
res = lm(value ~ group, data=df)
summary(res)
```

```

Call:
lm(formula = value ~ group, data = df)

Residuals:
    Min      1Q  Median      3Q     Max 
-1.9723 -0.5600  0.2511  0.5252  1.3889 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)   
(Intercept)   0.7039     0.3094   2.275  0.03538 * 
groupg2      -1.5008     0.4376  -3.430  0.00299 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.9785 on 18 degrees of freedom
Multiple R-squared:  0.3952,	Adjusted R-squared:  0.3616 
F-statistic: 11.76 on 1 and 18 DF,  p-value: 0.002989
```

14.6. Power calculations

The power of a statistical test is the probability that the test will reject the null hypothesis when the alternative hypothesis is true. In other words, the power of a statistical test is the probability of not making a Type II error. The power of a statistical test depends on the significance level (alpha), the sample size, and the effect size.

The `power.t.test` function can be used to calculate the power of a one-sample t-test.

Looking at `help("power.t.test")`, we see that the function takes the following arguments:

- `n` - sample size
- `delta` - effect size
- `sd` - standard deviation of the sample
- `sig.level` - significance level
- `power` - power

We need to supply four of these arguments to calculate the fifth. For example, if we want to calculate the power of a one-sample t-test with a sample size of 5, a standard deviation of 1, and an effect size of 1, we can use the following command:

```
power.t.test(n = 5, delta = 1, sd = 1, sig.level = 0.05)
```

```

     Two-sample t test power calculation 

              n = 5
          delta = 1
             sd = 1
      sig.level = 0.05
          power = 0.2859276
    alternative = two.sided

NOTE: n is number in *each* group
```

This gives a nice summary of the power calculation. We can also extract the power value from the result:

```r
power.t.test(n = 5, delta = 1, sd = 1,
             sig.level = 0.05, type='one.sample')$power
```

```
[1] 0.4013203
```

> **Tip**
>
> When getting results from a function that don't look "computable" such as those from `power.t.test`, you can use the `$` operator to extract the value you want. In this case, we want the `power` value from the result of `power.t.test`.
> How would you know what to extract? You can use the `names` function or the `str` function to see the structure of the result. For example:
>
> ```r
> names(power.t.test(n = 5, delta = 1, sd = 1,
>              sig.level = 0.05, type='one.sample'))
> ```
>
> ```
> [1] "n"           "delta"       "sd"          "sig.level"   "power"
> [6] "alternative" "note"        "method"
> ```
>
> ```r
> # or
> str(power.t.test(n = 5, delta = 1, sd = 1,
>              sig.level = 0.05, type='one.sample'))
> ```
>
> ```
> List of 8
>  $ n          : num 5
>  $ delta      : num 1
>  $ sd         : num 1
>  $ sig.level  : num 0.05
>  $ power      : num 0.401
>  $ alternative: chr "two.sided"
>  $ note       : NULL
>  $ method     : chr "One-sample t test power calculation"
>  - attr(*, "class")= chr "power.htest"
> ```

Alternatively, we may know a lot about our experimental system and want to calculate the sample size needed to achieve a certain power. For example, if we want to achieve a power of 0.8 with a standard deviation of 1 and an effect size of 1, we can use the following command:

```
power.t.test(delta = 1, sd = 1, sig.level = 0.05, power = 0.8, type = "one.sample")
```

```

     One-sample t test power calculation 

              n = 9.937864
          delta = 1
             sd = 1
      sig.level = 0.05
          power = 0.8
    alternative = two.sided
```

The power.t.test function is convenient and quite fast. As we've seen before, though, sometimes the distribution of the test statistics is now easily calculated. In those cases, we can use simulation to calculate the power of a statistical test. For example, if we want to calculate the power of a one-sample t-test with a sample size of 5, a standard deviation of 1, and an effect size of 1, we can use the following command:

```
sim_t_test_pval <- function(n = 5, delta = 1, sd = 1, sig.level = 0.05) {
    x = rnorm(n, delta, sd)
    t.test(x)$p.value <= sig.level
}
pow = mean(replicate(1000, sim_t_test_pval()))
pow
```

```
[1] 0.405
```

Let's break this down. First, we define a function called `sim_t_test_pval` that takes the same arguments as the `power.t.test` function. Inside the function, we simulate a sample of size `n` from a normal distribution with mean `delta` and standard deviation `sd`. Then, we perform a one-sample t-test on the sample and return a logical value indicating whether the p-value is less than the significance level. Next, we use the `replicate` function to repeat the simulation 1000 times. Finally, we calculate the proportion of simulations in which the p-value was less than the significance level. This proportion is an estimate of the power of the one-sample t-test.

Let's compare the results of the `power.t.test` function and our simulation-based approach:

```
power.t.test(n = 5, delta = 1, sd = 1, sig.level = 0.05, type='one.sample')$power
```

```
[1] 0.4013203
```

```
mean(replicate(1000, sim_t_test_pval(n = 5, delta = 1, sd = 1, sig.level = 0.05)))
```

```
[1] 0.414
```

14.7. Resources

See the pwr package for more information on power calculations.

15. K-means clustering

15.1. History of the k-means algorithm

The k-means clustering algorithm was first proposed by Stuart Lloyd in 1957 as a technique for pulse-code modulation. However, it was not published until 1982. In 1965, Edward W. Forgy published an essentially identical method, which became widely known as the k-means algorithm. Since then, k-means clustering has become one of the most popular unsupervised learning techniques in data analysis and machine learning.

K-means clustering is a method for finding patterns or groups in a dataset. It is an unsupervised learning technique, meaning that it doesn't rely on previously labeled data for training. Instead, it identifies structures or patterns directly from the data based on the similarity between data points (see Figure 15.1).

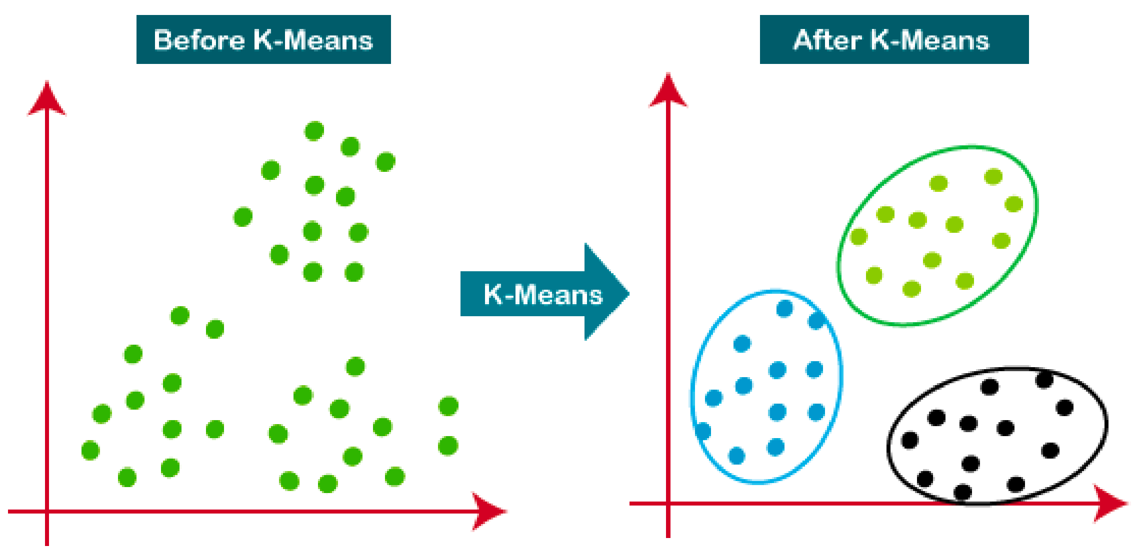


Figure 15.1.: K-means clustering takes a dataset and divides it into k clusters.

In simple terms, k-means clustering aims to divide a dataset into k distinct groups or clusters, where each data point belongs to the cluster with the nearest mean (average).

The goal is to minimize the variability within each cluster while maximizing the differences between clusters. This helps to reveal hidden patterns or relationships in the data that might not be apparent otherwise.

15.2. The k-means algorithm

The k-means algorithm follows these general steps:

1. Choose the number of clusters k.
2. Initialize the cluster centroids randomly by selecting k data points from the dataset.
3. Assign each data point to the nearest centroid.
4. Update the centroids by computing the mean of all the data points assigned to each centroid.
5. Repeat steps 3 and 4 until the centroids no longer change or a certain stopping criterion is met (e.g., a maximum number of iterations).

The algorithm converges when the centroids stabilize or no longer change significantly. The final clusters represent the underlying patterns or structures in the data. Advantages and disadvantages of k-means clustering

15.3. Pros and cons of k-means clustering

Compared to other clustering algorithms, k-means has several advantages:

- **Simplicity and ease of implementation** The k-means algorithm is relatively straightforward and can be easily implemented, even for large datasets.
- **Scalability** The algorithm can be adapted for large datasets using various optimization techniques or parallel processing.
- **Speed** K-means is generally faster than other clustering algorithms, especially when the number of clusters k is small.
- **Interpretability** The results of k-means clustering are easy to understand, as the algorithm assigns each data point to a specific cluster based on its similarity to the cluster's centroid.

However, k-means clustering has several disadvantages as well:

- **Choice of k** Selecting the appropriate number of clusters can be challenging and often requires domain knowledge or experimentation. A poor choice of k may yield poor results.
- **Sensitivity to initial conditions** The algorithm's results can vary depending on the initial placement of centroids. To overcome this issue, the algorithm can be run multiple times with different initializations and the best solution can be chosen based on a criterion (e.g., minimizing within-cluster variation).
- **Assumes spherical clusters** K-means assumes that clusters are spherical and evenly sized, which may not always be the case in real-world datasets. This can lead to poor performance if the underlying clusters have different shapes or densities.
- **Sensitivity to outliers** The algorithm is sensitive to outliers, which can heavily influence the position of centroids and the final clustering result. Preprocessing the data to remove or mitigate the impact of outliers can help improve the performance of k-means clustering.

Despite limitations, k-means clustering remains a popular and widely used method for exploring and analyzing data, particularly in biological data analysis, where identifying patterns and relationships can provide valuable insights into complex systems and processes.

15.4. An example of k-means clustering

15.4.1. The data and experimental background

The data we are going to use are from DeRisi, Iyer, and Brown (1997). From their abstract:

> DNA microarrays containing virtually every gene of Saccharomyces cerevisiae were used to carry out a comprehensive investigation of the temporal program of gene expression accompanying the metabolic shift from fermentation to respiration. The expression profiles observed for genes with known metabolic functions pointed to features of the metabolic reprogramming that occur during the diauxic shift, and the expression patterns of many previously uncharacterized genes provided clues to their possible functions.

These data are available from NCBI GEO as GSE28.

In the case of the baker's or brewer's yeast Saccharomyces cerevisiae growing on glucose with plenty of aeration, the diauxic growth pattern is commonly observed in batch culture.

During the first growth phase, when there is plenty of glucose and oxygen available, the yeast cells prefer glucose fermentation to aerobic respiration even though aerobic respiration is the more efficient pathway to grow on glucose. This experiment profiles gene expression for 6400 genes over a time course during which the cells are undergoing a diauxic shift.

The data in deRisi et al. have no replicates and are time course data. Sometimes, seeing how groups of genes behave can give biological insight into the experimental system or the function of individual genes. We can use clustering to group genes that have a similar expression pattern over time and then potentially look at the genes that do so.

Our goal, then, is to use `kmeans` clustering to divide highly variable (informative) genes into groups and then to visualize those groups.

15.5. Getting data

These data were deposited at NCBI GEO back in 2002. GEOquery can pull them out easily.

```
library(GEOquery)
gse = getGEO("GSE28")[[1]]
class(gse)
```

```
[1] "ExpressionSet"
attr(,"package")
[1] "Biobase"
```

GEOquery is a little dated and was written before the SummarizedExperiment existed. However, Bioconductor makes a conversion from the old ExpressionSet that GEOquery uses to the SummarizedExperiment that we see so commonly used now.

```
library(SummarizedExperiment)
gse = as(gse, "SummarizedExperiment")
gse
```

```
class: SummarizedExperiment
dim: 6400 7
metadata(3): experimentData annotation protocolData
assays(1): exprs
rownames(6400): 1 2 ... 6399 6400
```

```
rowData names(20): ID ORF ... FAILED IS_CONTAMINATED
colnames(7): GSM887 GSM888 ... GSM892 GSM893
colData names(33): title geo_accession ... supplementary_file
  data_row_count
```

Taking a quick look at the `colData()`, it might be that we want to reorder the columns a bit.

```
colData(gse)$title
```

```
[1] "diauxic shift timecourse: 15.5 hr" "diauxic shift timecourse: 0 hr"
[3] "diauxic shift timecourse: 18.5 hr" "diauxic shift timecourse: 9.5 hr"
[5] "diauxic shift timecourse: 11.5 hr" "diauxic shift timecourse: 13.5 hr"
[7] "diauxic shift timecourse: 20.5 hr"
```

So, we can reorder by hand to get the time course correct:

```
gse = gse[, c(2,4,5,6,1,3,7)]
```

15.6. Preprocessing

In gene expression data analysis, the primary objective is often to identify genes that exhibit significant differences in expression levels across various conditions, such as diseased vs. healthy samples or different time points in a time-course experiment. However, gene expression datasets are typically large, noisy, and contain numerous genes that do not exhibit substantial changes in expression levels. Analyzing all genes in the dataset can be computationally intensive and may introduce noise or false positives in the results.

One common approach to reduce the complexity of the dataset and focus on the most informative genes is to subset the genes based on their standard deviation in expression levels across the samples. The standard deviation is a measure of dispersion or variability in the data, and genes with high standard deviations have more variation in their expression levels across the samples.

By selecting genes with high standard deviations, we focus on genes that show relatively large changes in expression levels across different conditions. These genes are more likely to be biologically relevant and involved in the underlying processes or pathways of interest. In contrast, genes with low standard deviations exhibit little or no change in expression levels and are less likely to be informative for the analysis. It turns out that applying filtering based on criteria such as standard deviation can also increase power and reduce false positives in the analysis (Bourgon, Gentleman, and Huber 2010).

To subset the genes for analysis based on their standard deviation, the following steps can be followed: Calculate the standard deviation of each gene's expression levels across all samples. Set a threshold for the standard deviation, which can be determined based on domain knowledge, data distribution, or a specific percentile of the standard deviation values (e.g., selecting the top 10% or 25% of genes with the highest standard deviations). Retain only the genes with a standard deviation above the chosen threshold for further analysis.

By subsetting the genes based on their standard deviation, we can reduce the complexity of the dataset, speed up the subsequent analysis, and increase the likelihood of detecting biologically meaningful patterns and relationships in the gene expression data. The threshold for the standard deviation cutoff is rather arbitrary, so it may be beneficial to try a few to check for sensitivity of findings.

```
sds = apply(assays(gse)[[1]], 1, sd)
hist(sds)
```

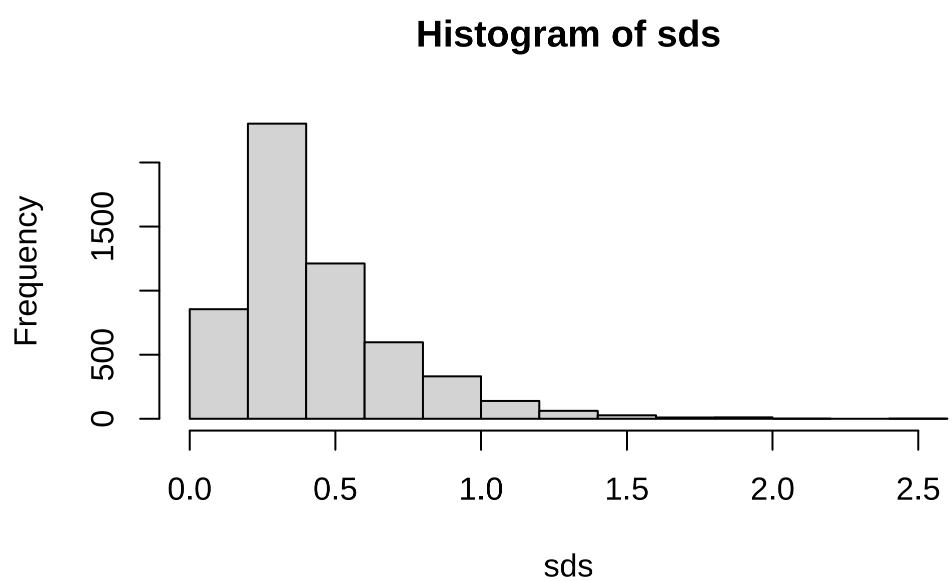


Figure 15.2.: Histogram of standard deviations for all genes in the deRisi dataset.

Examining the plot, we can see that the most highly variable genes have an sd > 0.8 or so (arbitrary). We can, for convenience, create a new `SummarizedExperiment` that contains only our most highly variable genes.

```
idx = sds>0.8 & !is.na(sds)
gse_sub = gse[idx,]
```

15.7. Clustering

Now, `gse_sub` contains a subset of our data.

The `kmeans` function takes a matrix and the number of clusters as arguments.

```
k = 4
km = kmeans(assays(gse_sub)[[1]], 4)
```

The `km` kmeans result contains a vector, `km$cluster`, which gives the cluster associated with each gene. We can plot the genes for each cluster to see how these different genes behave.

```
expression_values = assays(gse_sub)[[1]]
par(mfrow=c(2,2), mar=c(3,4,1,2)) # this allows multiple plots per page
for(i in 1:k) {
    matplot(t(expression_values[km$cluster==i, ]), type='l', ylim=c(-3,3),
            ylab = paste("cluster", i))
}
```

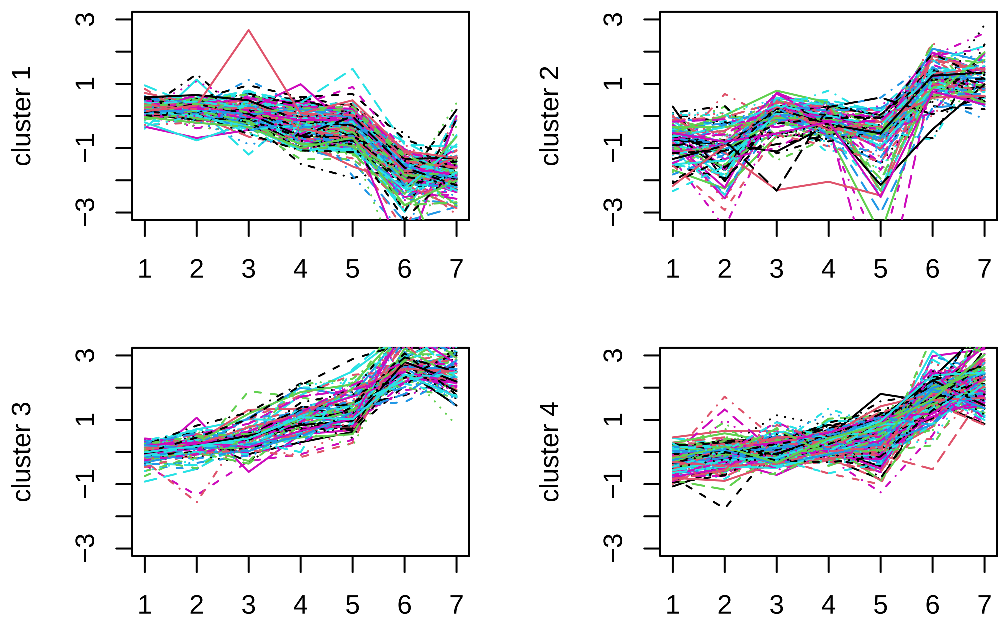


Figure 15.3.: Gene expression profiles for the four clusters identified by k-means clustering. Each line represents a gene in the cluster, and each column represents a time point in the experiment. Each cluster shows a distinct trend where the genes in the cluster are potentially co-regulated.

Try this with different size k. Perhaps go back to choose more genes (using a smaller cutoff for sd).

15.8. Summary

In this lesson, we have learned how to use k-means clustering to identify groups of genes that behave similarly over time. We have also learned how to subset our data to focus on the most informative genes.

16. Machine Learning

16.1. What is Machine Learning?

Machine learning is a subfield of artificial intelligence that focuses on the development of algorithms and models that enable computers to learn and make decisions or predictions without explicit programming. It has emerged as a powerful tool for solving complex problems across various industries, including healthcare, finance, marketing, and natural language processing. This chapter provides an overview of machine learning, its types, key concepts, applications, and challenges.

Machine learning in biology is a really broad topic. Greener et al. (2022) present a nice overview of the different types of machine learning methods that are used in biology. Libbrecht and Noble (2015) also present an early review of machine learning in genetics and genomics.

16.2. Classes of Machine Learning

16.2.1. Supervised learning

Supervised learning is a type of machine learning where the model learns from labeled data, i.e., input-output pairs, to make predictions. It includes tasks like regression (predicting continuous values) and classification (predicting discrete classes or categories).

16.2.2. Unsupervised learning

Unsupervised learning involves learning from unlabeled data, where the model discovers patterns or structures within the data. Common unsupervised learning tasks include clustering (grouping similar data points), dimensionality reduction (reducing the number of features or variables), and anomaly detection (identifying unusual data points).

Terminology and Concepts

- **Data** Data is the foundation of machine learning and can be structured (tabular) or unstructured (text, images, audio). It is usually divided into training, validation, and testing sets for model development and evaluation.
- **Features** Features are the variables or attributes used to describe the data points. Feature engineering and selection are crucial steps in machine learning to improve model performance and interpretability.
- **Models and Algorithms** Models are mathematical representations of the relationship between features and the target variable(s). Algorithms are the methods used to train models, such as linear regression, decision trees, and neaural networks.
- **Hyperparameters and Tuning** Hyperparameters are adjustable parameters that control the learning process of an algorithm. Tuning involves finding the optimal set of hyperparameters to improve model performance.
- **Evaluation Metrics** Evaluation metrics quantify the performance of a model, such as accuracy, precision, recall, F1-score (for classification), and mean squared error, R-squared (for regression).

```r
set.seed(123)
sinsim <- function(n,sd=0.1) {
  x <- seq(0,1,length.out=n)
  y <- sin(2*pi*x) + rnorm(n,0,sd)
  return(data.frame(x=x,y=y))
}
dat <- sinsim(100,0.25)
library(ggplot2)
library(patchwork)
p_base <- ggplot(dat,aes(x=x,y=y)) +
 geom_point(alpha=0.7) +
 theme_bw()
p_lm <- p_base +
 geom_smooth(method="lm", se=FALSE, alpha=0.6, formula = y ~ x)
p_lmsin <- p_base +
 geom_smooth(method="lm",formula=y~sin(2*pi*x), se=FALSE, alpha=0.6)
p_loess_wide <- p_base +
  geom_smooth(method="loess",span=0.5, se=FALSE, alpha=0.6, formula = y ~ x)
```

```
p_loess_narrow <- p_base +
 geom_smooth(method="loess",span=0.1, se=FALSE, alpha=0.6, formula = y ~ x)
p_lm + p_lmsin + p_loess_wide + p_loess_narrow + plot_layout(ncol=2) +
  plot_annotation(tag_levels = 'A') &
  theme(plot.tag = element_text(size = 8))
```

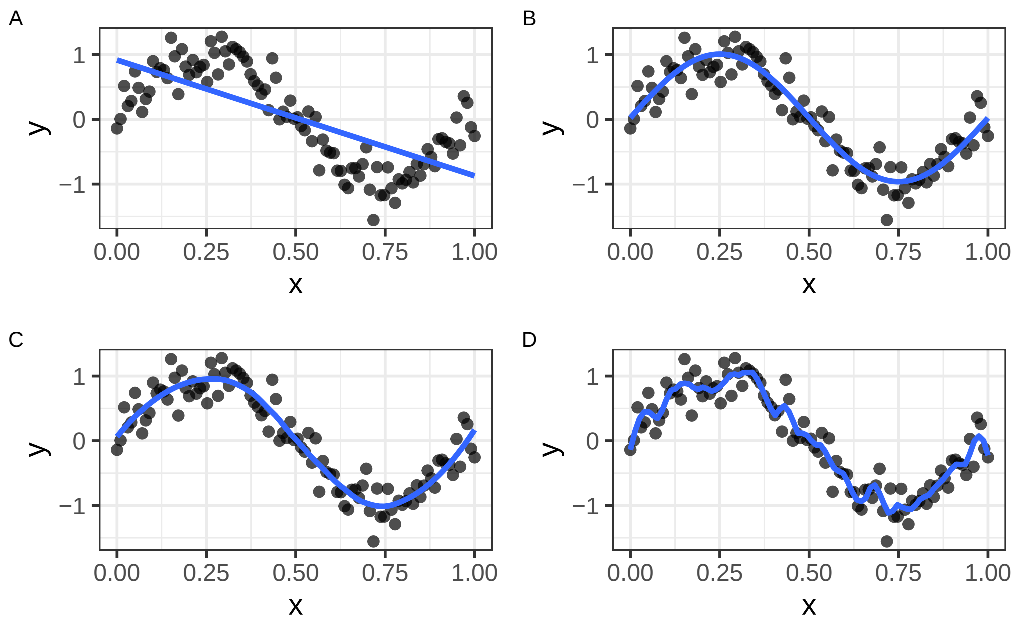


Figure 16.1.: Data simulated according to the function $f(x) = sin(2\pi x) + N(0, 0.25)$ fitted with four different models. A) A simple linear model demonstrates *underfitting*. B) A linear model with a sin function ($y = sin(2\pi x)$) and C) a loess model with a wide span (0.5) demonstrate *good fits*. D) A loess model with a narrow span (0.1) is a good example of *overfitting*.

In Figure 16.1, we simulate data according to the function $f(x) = sin(2\pi x) + N(0, 0.25)$ and fit four different models. Choosing a model that is too simple (A) will result in *underfitting* the data, while choosing a model that is too complex (D) will result in *overfitting* the data.

When thinking about machine learning, it can help to have a simple framework in mind. In Figure 16.2, we present a simple view of machine learning according to the scikit-learn package.

We're going to focus on supervised learning here. Here is a rough schematic (see Figure 16.3) of the supervised learning process from the mlr3 book.

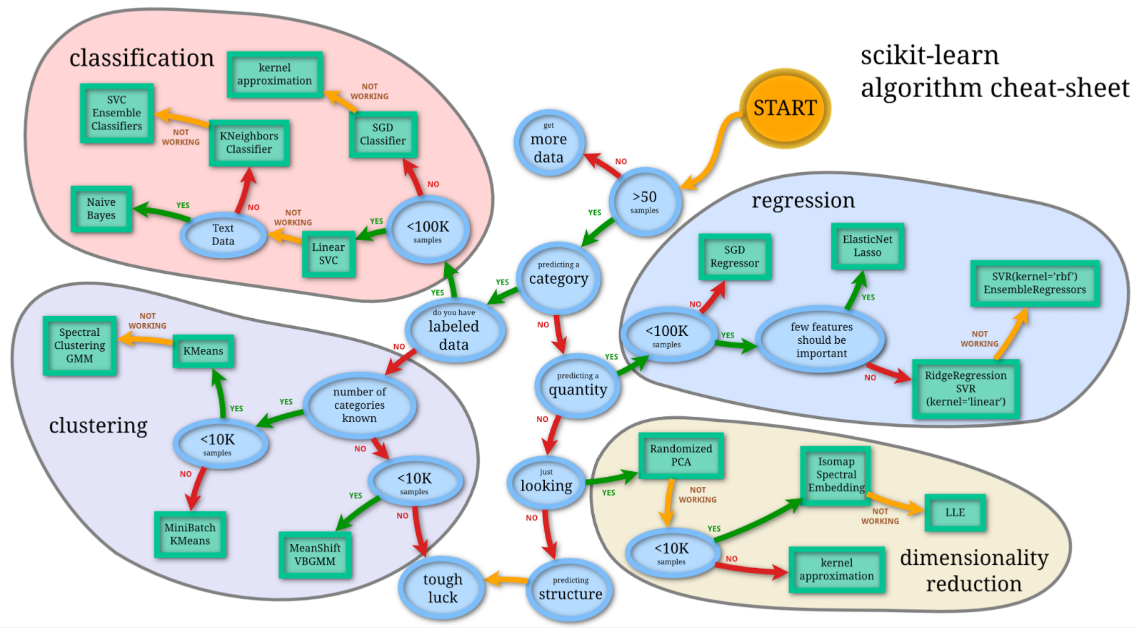


Figure 16.2.: A simple view of machine learning according the sklearn.

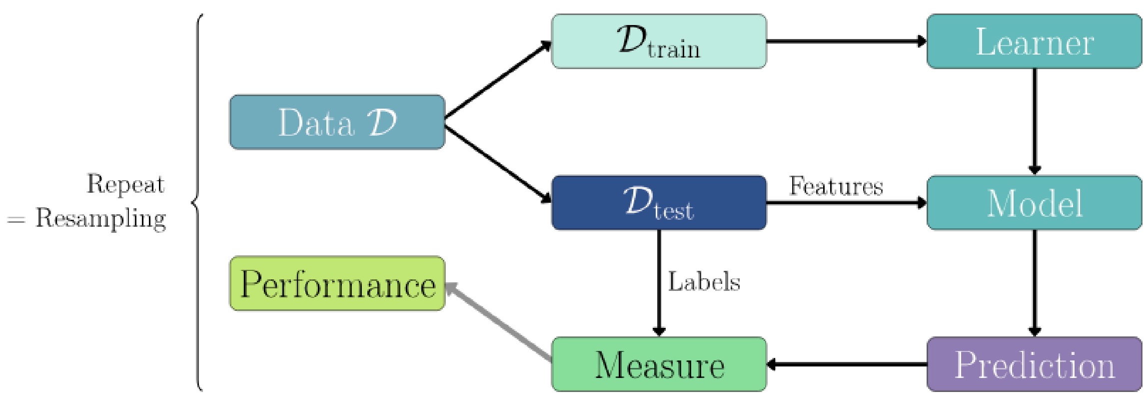


Figure 16.3.: A schematic of the supervised learning process.

In nearly all cases, we will have a training set and a test set. The training set is used to train the model, and the test set is used to evaluate the model (see Figure 16.4). Even when we don't have a separate test set, we will usually create one by splitting the data.

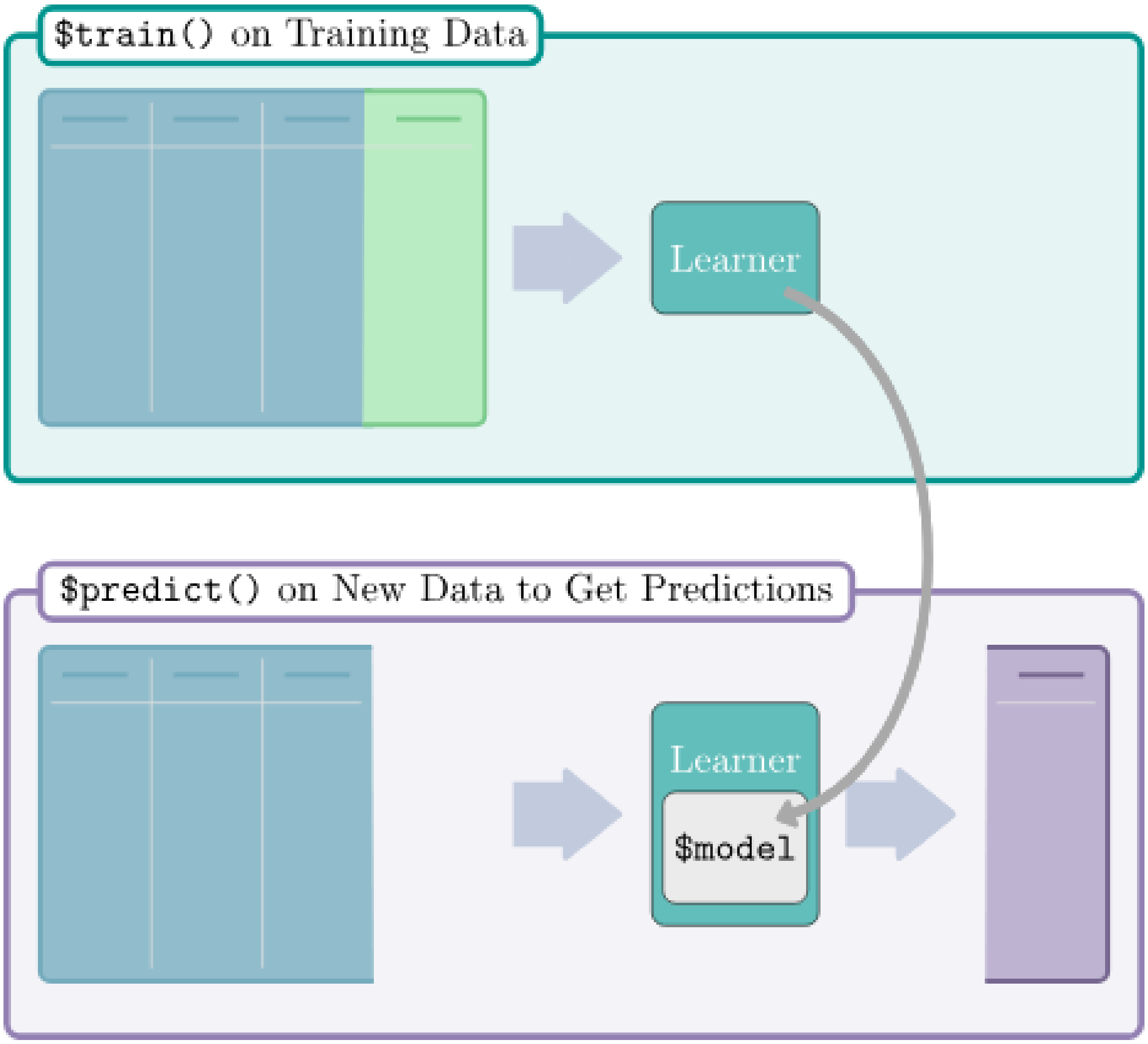


Figure 16.4.: Training and testing sets.

16.3. Supervised Learning

16.3.1. Linear regression

In statistics, **linear regression** is a linear approach for modelling the relationship between a scalar response and one or more explanatory variables (also known as dependent and independent variables). The case of one explanatory variable is called *simple linear regression*; for more than one, the process is called **multiple linear regression**. This term is distinct from multivariate linear regression, where multiple correlated dependent variables are predicted, rather than a single scalar variable.

In linear regression, the relationships are modeled using linear predictor functions whose unknown model parameters are estimated from the data. Such models are called linear models. Most commonly, the conditional mean of the response given the values of the explanatory variables (or predictors) is assumed to be an affine function of those values; less commonly, the conditional median or some other quantile is used. Like all forms of regression analysis, linear regression focuses on the conditional probability distribution of the response given the values of the predictors, rather than on the joint probability distribution of all of these variables, which is the domain of multivariate analysis.

Linear regression was the first type of regression analysis to be studied rigorously, and to be used extensively in practical applications. This is because models which depend linearly on their unknown parameters are easier to fit than models which are non-linearly related to their parameters and because the statistical properties of the resulting estimators are easier to determine.

16.3.2. K-nearest Neighbor

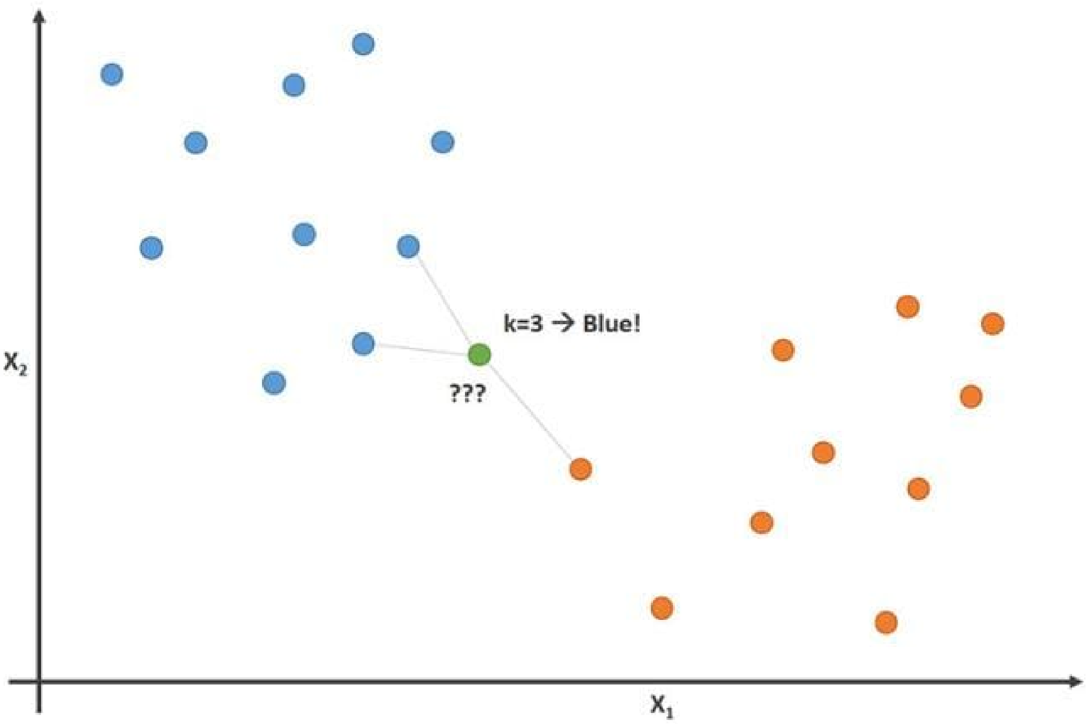


Figure 16.5.: **Figure**. The k-nearest neighbor algorithm can be used for regression or classification.

The ***k*-nearest neighbors algorithm** (***k*-NN**) is a non-parametric supervised learning method first developed by Evelyn Fix and Joseph Hodges in 1951, and later expanded by Thomas Cover. It is used for classification and regression. In both cases, the input consists of the k closest training examples in a data set.

The k-nearest neighbor (k-NN) algorithm is a simple, yet powerful, supervised machine learning method used for classification and regression tasks. It is an instance-based, non-parametric learning method that stores the entire training dataset and makes predictions based on the similarity between data points. The underlying principle of the k-NN algorithm is that similar data points (those that are close to each other in multidimensional space) are likely to have similar outcomes or belong to the same class.

Here's a description of how the k-NN algorithm works:

1. Determine the value of k: The first step is to choose the number of nearest neighbors (k) to consider when making predictions. The value of k is a user-defined hyperparameter and can significantly impact the algorithm's performance. A small value of k can lead to overfitting, while a large value may result in underfitting.
2. Compute distance: Calculate the distance between the new data point (query point) and each data point in the training dataset. The most common distance metrics used are Euclidean, Manhattan, and Minkowski distance. The choice of distance metric depends on the problem and the nature of the data.
3. Find k-nearest neighbors: Identify the k data points in the training dataset that are closest to the query point, based on the chosen distance metric.
4. Make predictions: Once the k-nearest neighbors are identified, the final step is to make predictions. The prediction for the query point can be made in two ways:

    a. For classification, determine the class labels of the k-nearest neighbors and assign the class label with the highest frequency (majority vote) to the query point. In case of a tie, one can choose the class with the smallest average distance to the query point or randomly select one among the tied classes.
    b. For regression tasks, the k-NN algorithm follows a similar process, but instead of majority voting, it calculates the mean (or median) of the target values of the k-nearest neighbors and assigns it as the prediction for the query point.

The k-NN algorithm is known for its simplicity, ease of implementation, and ability to handle multi-class problems. However, it has some drawbacks, such as high computational cost (especially for large datasets), sensitivity to the choice of k and distance metric, and poor performance with high-dimensional or noisy data. Scaling and preprocessing the data, as well as using dimensionality reduction techniques, can help mitigate some of these issues.

- In *k-NN classification*, the output is a class membership. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically small). If $k = 1$, then the object is simply assigned to the class of that single nearest neighbor.
- In *k-NN regression*, the output is the property value for the object. This value is the average of the values of k nearest neighbors.

k-NN is a type of classification where the function is only approximated locally and all computation is deferred until function evaluation. Since this algorithm relies on distance for classification, if the features represent different physical units or come in vastly different scales then normalizing the training data can improve its accuracy dramatically.

Both for classification and regression, a useful technique can be to assign weights to the contributions of the neighbors, so that the nearer neighbors contribute more to the average than the more distant ones. For example, a common weighting scheme consists in giving each neighbor a weight of $1/d$, where d is the distance to the neighbor.

The neighbors are taken from a set of objects for which the class (for k-NN classification) or the object property value (for k-NN regression) is known. This can be thought of as the training set for the algorithm, though no explicit training step is required.

16.4. Penalized regression

Adapted from http://www.sthda.com/english/articles/37-model-selection-essentials-in-r/153-penalized-regression-essentials-ridge-lasso-elastic-net/.

Penalized regression is a type of regression analysis that introduces a penalty term to the loss function in order to prevent overfitting and improve the model's ability to generalize. Remember that in regression, the *loss* function is the sum of squares Equation 16.1.

$$L = \sum_{i=0}^{n} (\hat{y}_i - y_i)^2 \tag{16.1}$$

In Equation 16.1, $\hat{y}_i$ is the predicted output, y_i is the actual output, and n is the number of observations. The goal of regression is to minimize the loss function by finding the optimal values of the model parameters or coefficients. The model parameters are estimated using the training data. The model is then evaluated using the test data. If the model performs well on the training data but poorly on the test data, it is said to be overfit. Overfitting occurs when the model learns the training data too well, including the noise, and is not able to generalize well to new data. This is a common problem in machine learning, particularly when there are a large number of predictors compared to the number of observations, and can be addressed by penalized regression.

The two most common types of penalized regression are Ridge Regression (L2 penalty) and LASSO Regression (L1 penalty). Both Ridge and LASSO help to reduce model complexity and prevent over-fitting which may result from simple linear regression. However, the choice between Ridge and LASSO depends on the situation and the dataset at hand. If feature selection is important for the interpretation of the model, LASSO might be preferred. If the goal is prediction accuracy and the model needs to retain all features, Ridge might be the better choice.

16.4.1. Ridge regression

Ridge regression shrinks the regression coefficients, so that variables, with minor contribution to the outcome, have their coefficients close to zero. The shrinkage of the coefficients is achieved by penalizing the regression model with a penalty term called L2-norm, which is the sum of the squared coefficients. The amount of the penalty can be fine-tuned using a constant called lambda ( ). Selecting a good value for is critical. When =0, the penalty term has no effect, and ridge regression will produce the classical least square coefficients. However, as increases to infinite, the impact of the shrinkage penalty grows, and the ridge regression coefficients will get close zero. The loss function for Ridge Regression is:

$$L = \sum_{i=0}^{n} \left(\hat{y}_i - y_i\right)^2 + \lambda \sum_{j=0}^{k} \beta_j^2 \tag{16.2}$$

Here, $\hat{y}_i$ is the predicted output, y_i is the actual output, β_j represents the model parameters or coefficients, and is the regularization parameter. The second term, j^2, is the penalty term where all parameters are squared and summed. Ridge regression tends to shrink the coefficients but doesn't necessarily zero them.

Note that, in contrast to the ordinary least square regression, ridge regression is highly affected by the scale of the predictors. Therefore, it is better to standardize (i.e., scale) the predictors before applying the ridge regression (James et al. 2014), so that all the predictors are on the same scale. The standardization of a predictor x, can be achieved using the formula $x' = \frac{x}{sd(x)}$, where $sd(x)$ is the standard deviation of x. The consequence of this is that, all standardized predictors will have a standard deviation of one allowing the final fit to not depend on the scale on which the predictors are measured.

One important advantage of the ridge regression, is that it still performs well, compared to the ordinary least square method (see Equation 16.1), in a situation where you have a large multivariate data with the number of predictors (p) larger than the number of observations (n). One disadvantage of the ridge regression is that, it will include all the predictors in the final model, unlike the stepwise regression methods, which will generally select models that involve a reduced set of variables. Ridge regression shrinks the coefficients towards zero, but it will not set any of them exactly to zero. The LASSO regression is an alternative that overcomes this drawback.

16.4.2. LASSO regression

LASSO stands for *Least Absolute Shrinkage and Selection Operator*. It shrinks the regression coefficients toward zero by penalizing the regression model with a penalty term called

L1-norm, which is the sum of the absolute coefficients. In the case of LASSO regression, the penalty has the effect of forcing some of the coefficient estimates, with a minor contribution to the model, to be exactly equal to zero. This means that, LASSO can be also seen as an alternative to the subset selection methods for performing variable selection in order to reduce the complexity of the model. As in ridge regression, selecting a good value of λ for the LASSO is critical. The loss function for LASSO Regression is:

$$L = \sum_{i=0}^{n} \left(\hat{y}_i - y_i\right)^2 + \lambda \sum_{j=0}^{k} |\beta_j| \tag{16.3}$$

Similar to Ridge, ŷi is the predicted output, yi is the actual output, j represents the model parameters or coefficients, and is the regularization parameter. The second term, | j|, is the penalty term where the absolute values of all parameters are summed. LASSO regression tends to shrink the coefficients and can zero out some of them, effectively performing variable selection.

One obvious advantage of LASSO regression over ridge regression, is that it produces simpler and more interpretable models that incorporate only a reduced set of the predictors. However, neither ridge regression nor the LASSO will universally dominate the other. Generally, LASSO might perform better in a situation where some of the predictors have large coefficients, and the remaining predictors have very small coefficients. Ridge regression will perform better when the outcome is a function of many predictors, all with coefficients of roughly equal size (James et al. 2014).

Cross-validation methods can be used for identifying which of these two techniques is better on a particular data set.

16.4.3. Elastic Net

Elastic Net produces a regression model that is penalized with both the L1-norm and L2-norm. The consequence of this is to effectively shrink coefficients (like in ridge regression) and to set some coefficients to zero (as in LASSO).

16.4.4. Classification and Regression Trees (CART)

Decision Tree Learning is supervised learning approach used in statistics, data mining and machine learning. In this formalism, a classification or regression decision tree is used as a predictive model to draw conclusions about a set of observations. Decision trees are a popular machine learning method used for both classification and regression tasks. They are hierarchical, tree-like structures that model the relationship between features and the target variable by recursively splitting the data into subsets based on the feature values. Each internal node in the tree represents a decision or test on a feature, and each branch represents the outcome of that test. The leaf nodes contain the final prediction, which is the majority class for classification tasks or the mean/median of the target values for regression tasks.

Here's an overview of the decision tree learning process:

- Select the best feature and split value: Start at the root node and choose the feature and split value that results in the maximum reduction of impurity (or increase in information gain) in the child nodes. For classification tasks, impurity measures like Gini index or entropy are commonly used, while for regression tasks, mean squared error (MSE) or mean absolute error (MAE) can be used.
- Split the data: Partition the dataset into subsets based on the chosen feature and split value.
- Recursion: Repeat steps 1 and 2 for each subset until a stopping criterion is met. Stopping criteria can include reaching a maximum tree depth, a minimum number of samples per leaf, or no further improvement in impurity.
- Prune the tree (optional): To reduce overfitting, decision trees can be pruned by removing branches that do not significantly improve the model's performance on the validation dataset. This can be done using techniques like reduced error pruning or cost-complexity pruning.

Decision trees have several advantages, such as:

- **Interpretability** They are easy to understand, visualize, and explain, even for non-experts.
- **Minimal data preprocessing** Decision trees can handle both numerical and categorical data, and they are robust to outliers and missing values.
- **Non-linear relationships** They can capture complex non-linear relationships between features and the target variable.

However, decision trees also have some drawbacks:

- **Overfitting** They are prone to overfitting, especially when the tree is deep or has few samples per leaf. Pruning and setting stopping criteria can help mitigate this issue.
- **Instability** Small changes in the data can lead to different tree structures. This can be addressed by using ensemble methods like random forests or gradient boosting machines (GBMs).

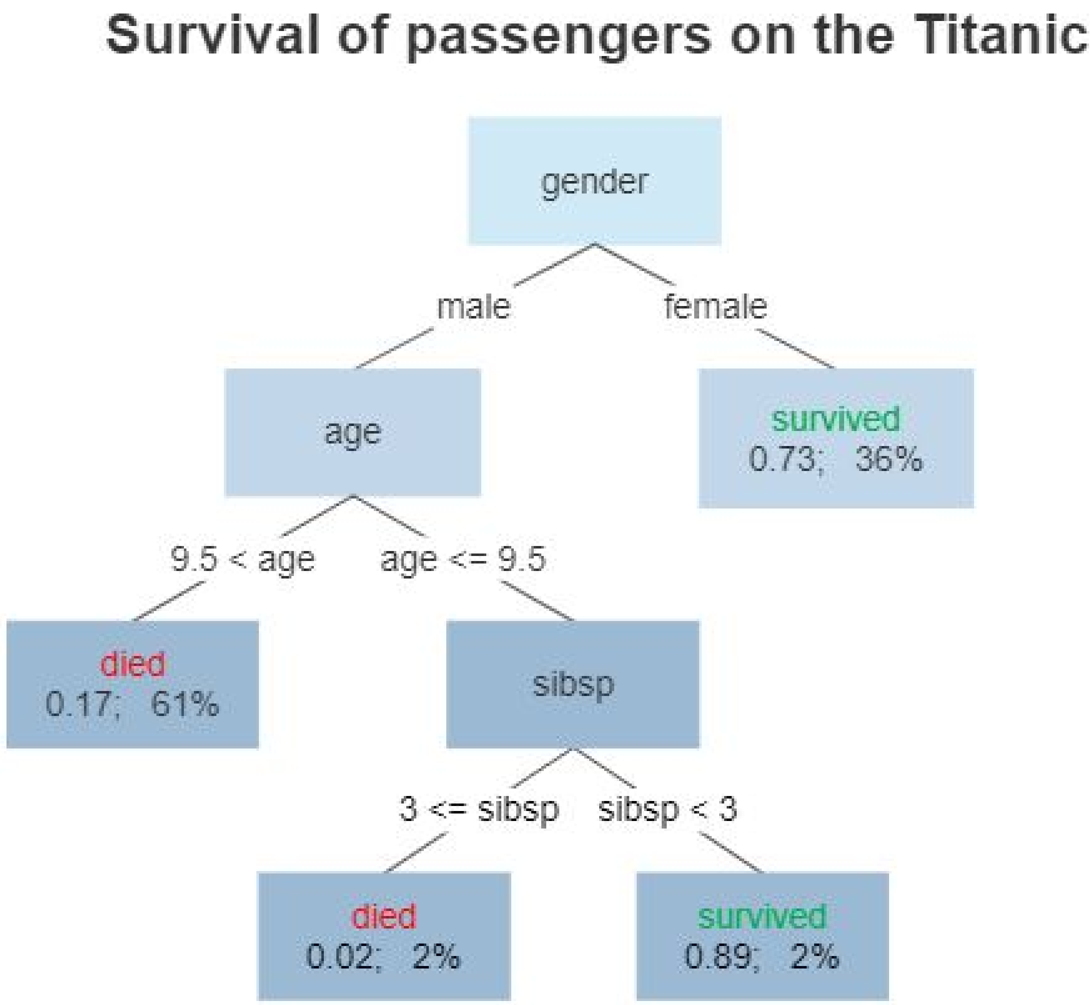


Figure 16.6.: An example of a decision tree that performs classification, also sometimes called a classification tree.

- **Greedy learning** Decision tree algorithms use a greedy approach, meaning they make locally optimal choices at each node. This may not always result in a globally optimal tree.

Despite these limitations, decision trees are widely used in various applications due to their simplicity, interpretability, and ability to handle diverse data types.

16.4.5. RandomForest

Random forests or **random decision forests** is an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time. For classification tasks, the output of the random forest is the class selected by most trees. For regression tasks, the mean or average prediction of the individual trees is returned. Random decision forests correct for decision trees' habit of overfitting to their training set. Random forests generally outperform decision trees, but their accuracy is lower than gradient boosted trees[*citation needed*]. However, data characteristics can affect their performance.

The first algorithm for random decision forests was created in 1995 by Tin Kam Ho using the random subspace method, which, in Ho's formulation, is a way to implement the "stochastic discrimination" approach to classification proposed by Eugene Kleinberg.

An extension of the algorithm was developed by Leo Breiman and Adele Cutler, who registered "Random Forests" as a trademark in 2006 (as of 2019[update], owned by Minitab, Inc.). The extension combines Breiman's "bagging" idea and random selection of features, introduced first by Ho and later independently by Amit and Geman in order to construct a collection of decision trees with controlled variance.

Random forests are frequently used as "blackbox" models in businesses, as they generate reasonable predictions across a wide range of data while requiring little configuration.

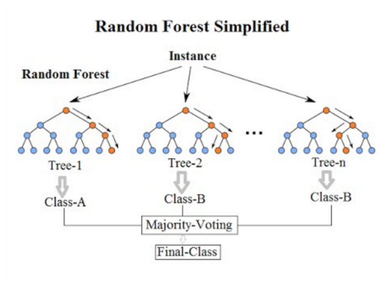


Figure 16.7.: Random forests or random decision forests is an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time.

17. Machine Learning 2

17.1. Overview

In this chapter, we focus on practical aspects of machine learning. The goal is to provide a hands-on introduction to the application of machine learning techniques to real-world data. While the theoretical foundations of machine learning are important, the ability to apply these techniques to solve practical problems is equally crucial. In this chapter, we will use the `mlr3` package in R to build and evaluate machine learning models for classification and regression tasks.

We will use three examples to illustrate the machine learning workflow:

1. **Cancer types classification**: We will classify different types of cancer based on gene expression data.
2. **Age prediction from DNA methylation**: We will predict the chronological age of individuals based on DNA methylation patterns.
3. **Gene expression prediction**: We will predict gene expression levels based on histone modification data.

We'll be applying knn, decision trees, and random forests, linear regression, and penalized regression models to these datasets.

The mlr3 R package is a modern, object-oriented machine learning framework in R that builds on the success of its predecessor, the mlr package. It provides a flexible and extensible platform for handling common machine learning tasks such as data preprocessing, model training, hyperparameter tuning, and model evaluation Figure 17.1. The package is designed to guide and standardize the process of using complex machine learning pipelines.

17.1.1. Key features of mlr3

- **Task abstraction** mlr3 encapsulates different types of learning problems like classification, regression, and survival analysis into "Task" objects, making it easier to

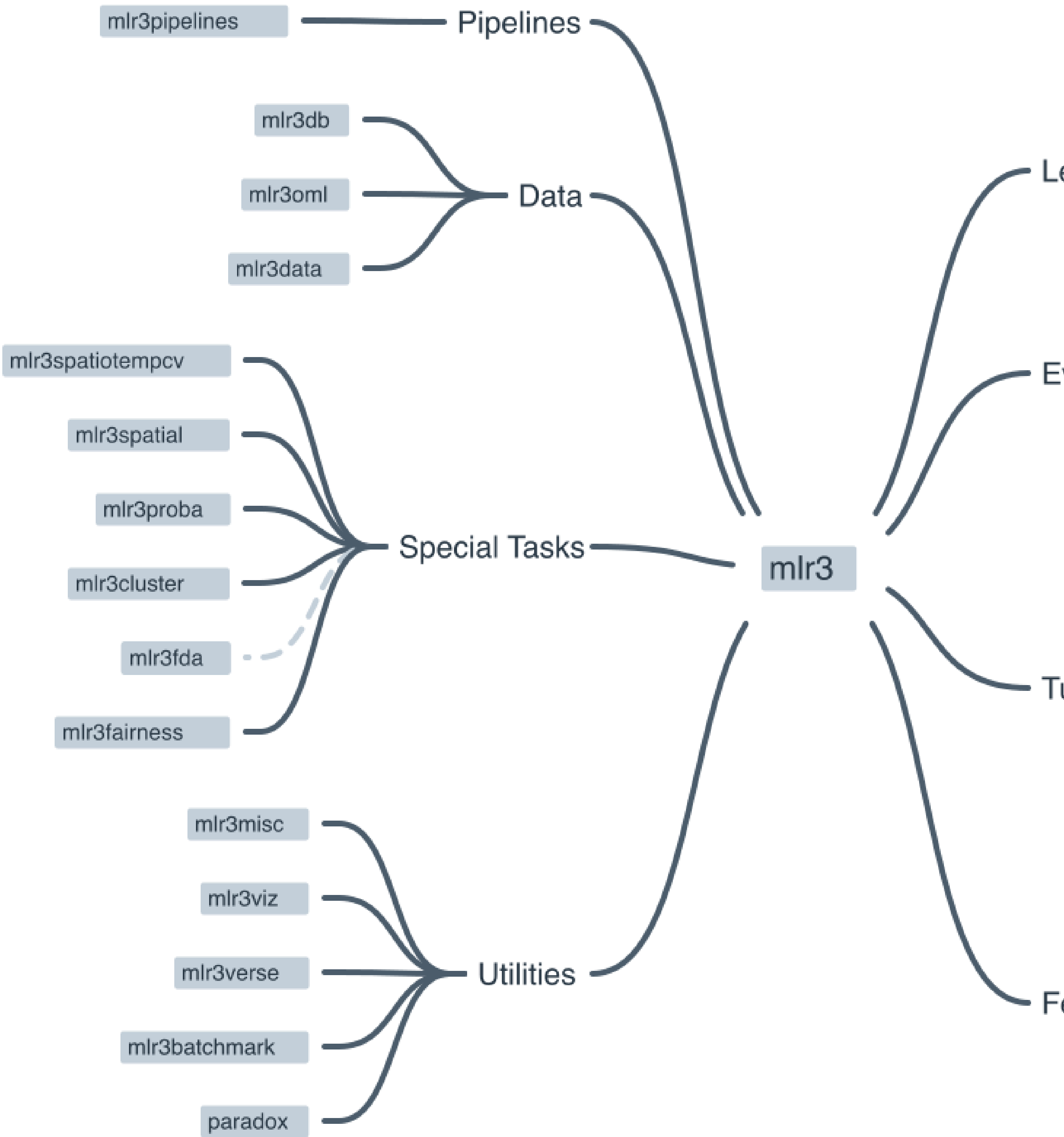


Figure 17.1.: The mlr3 ecosystem.

handle various learning scenarios. Examples of tasks include classification tasks, regression tasks, and survival tasks.

- **Modular design** The package follows a modular design, allowing users to quickly swap out different components such as learners (algorithms), measures (performance metrics), and resampling strategies. Examples of learners include linear regression, logistic regression, and random forests. Examples of measures include accuracy, precision, recall, and F1 score. Examples of resampling strategies include cross-validation, bootstrapping, and holdout validation.
- **Extensibility** Users can extend the functionality of mlr3 by adding custom components like learners, measures, and preprocessing steps via the R6 object-oriented system.
- **Preprocessing** mlr3 provides a flexible way to preprocess data using "PipeOps" (pipeline operations), allowing users to create reusable preprocessing pipelines.
- **Tuning and model selection** mlr3 supports hyperparameter tuning and model selection using various search strategies like grid search, random search, and Bayesian optimization.
- **Parallelization** The package allows for parallelization of model training and evaluation, making it suitable for large-scale machine learning tasks.
- **Benchmarking** mlr3 facilitates benchmarking of multiple algorithms on multiple tasks, simplifying the process of comparing and selecting the best models.

You can find more information, including tutorials and examples, on the official mlr3 GitHub repository[1] and the mlr3 book[2].

17.2. The mlr3 workflow

The mlr3 package is designed to simplify the process of creating and deploying complex machine learning pipelines. The package follows a modular design, which means that users can quickly swap out different components such as learners (algorithms), measures (performance metrics), and resampling strategies. The package also supports parallelization of model training and evaluation, making it suitable for large-scale machine learning tasks.

The following sections describe each of these steps in detail.

[1] https://github.com/mlr-org/mlr3
[2] https://mlr3book.mlr-org.com/

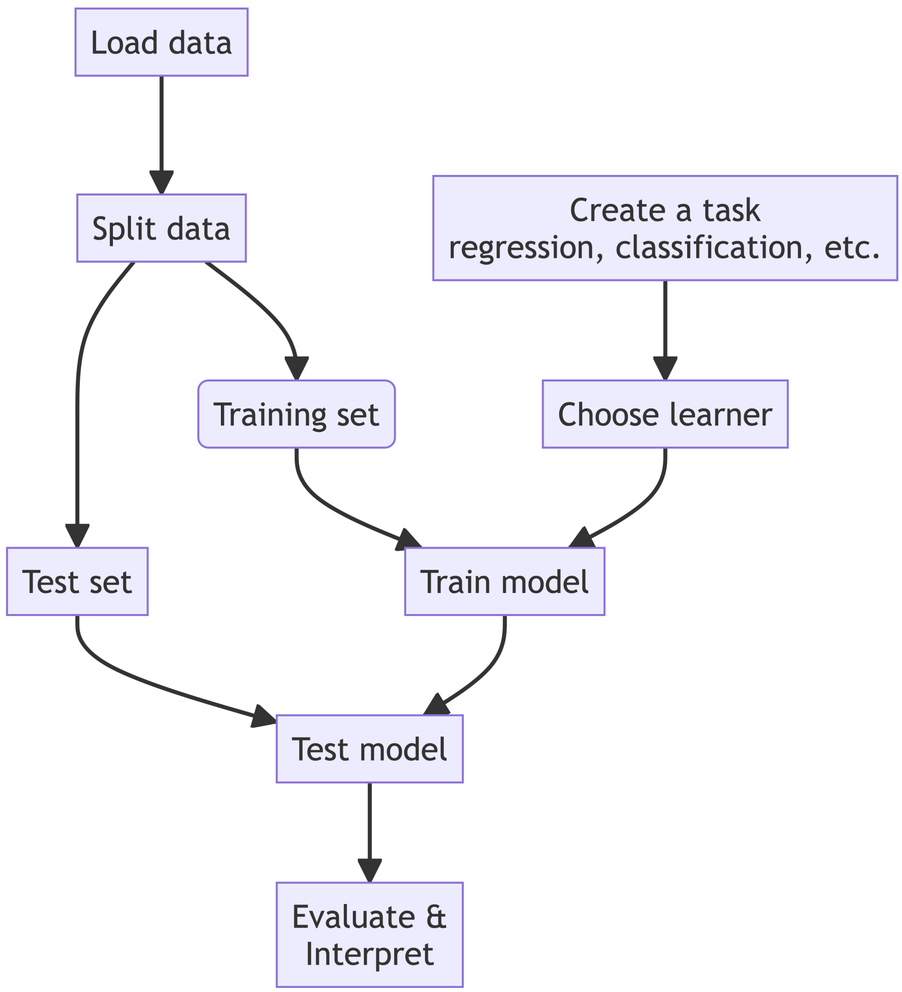


Figure 17.2.: The simplified workflow of a machine learning pipeline using mlr3.

17.2.1. The machine learning Task

Imagine you want to teach a computer how to make predictions or decisions, similar to how you might teach a student. To do this, you need to clearly define what you want the computer to learn and work on. This is called defining a "task." Let's break down what this involves and why it's important.

17.2.1.1. Step 1: Understand the Problem

First, think about what problem you want to solve or what question you want the computer to answer. For example: - Do you want to predict the weather for tomorrow? - Are you trying to figure out if an email is spam or not? - Do you want to know how much a house might sell for?

These questions define your **task type**. In machine learning, there are several common task types:

- **Classification:** Deciding which category something belongs to (e.g., spam or not spam).
- **Regression:** Predicting a number (e.g., the price of a house).
- **Clustering:** Grouping similar items together (e.g., customer segmentation).

17.2.1.2. Step 2: Choose Your Data

Next, you need data that is related to your problem. Think of data as the information or examples you'll use to teach the computer. For instance, if your task is to predict house prices, your data might include:

- The size of the house
- The number of bedrooms
- The location of the house
- The age of the house

These pieces of information are called **features**. Features are the input that the computer uses to make predictions.

17.2.1.3. Step 3: Define the Target

Along with features, you need to define the **target**. The target is what you want to predict or decide. In the house price example, the target would be the actual price of the house.

17.2.1.4. Step 4: Create the Task

Now that you have your problem, data, and target, you can create the task. In mlr3, a task brings together the type of problem (task type), the features (input data), and the target (what you want to predict).

Here's a simple summary:

1. **Task Type:** What kind of problem are you solving? (e.g., classification, regression)
2. **Features:** What information do you have to make the prediction? (e.g., size, location)
3. **Target:** What are you trying to predict? (e.g., house price)

By clearly defining these elements, you set a solid foundation for the machine learning process. This helps ensure that the computer can learn effectively and make accurate predictions.

17.2.1.5. mlr3 and Tasks

The mlr3 package uses the concept of "Tasks" to encapsulate different types of learning problems like classification, regression, and survival analysis. A Task contains the data (features and target variable) and additional metadata to define the machine learning problem. For example, in a classification task, the target variable is a label (stored as a character or factor), while in a regression task, the target variable is a numeric quantity (stored as an integer or numeric).

There are a number of Task Types that are supported by mlr3. To create a task from a `data.frame()`, `data.table()` or `Matrix()`, you first need to select the right task type:

- **Classification Task**: The target is a label (stored as `character` or `factor`) with only relatively few distinct values → `TaskClassif`.
- **Regression Task**: The target is a numeric quantity (stored as `integer` or `numeric`) → `TaskRegr`.
- **Survival Task**: The target is the (right-censored) time to an event. More censoring types are currently in development → `mlr3proba::TaskSurv` in add-on package mlr3proba.
- **Density Task**: An unsupervised task to estimate the density → `mlr3proba::TaskDens` in add-on package mlr3proba.

- **Cluster Task**: An unsupervised task type; there is no target and the aim is to identify similar groups within the feature space → `mlr3cluster::TaskClust` in add-on package mlr3cluster.
- **Spatial Task**: Observations in the task have spatio-temporal information (e.g. coordinates) → `mlr3spatiotempcv::TaskRegrST` or `mlr3spatiotempcv::TaskClassifST` in add-on package mlr3spatiotempcv.
- **Ordinal Regression Task**: The target is ordinal → `TaskOrdinal` in add-on package mlr3ordinal (still in development).

17.2.2. The "Learner" in Machine Learning

After you've defined your task, the next step in teaching a computer to make predictions or decisions is to choose a "learner." Let's explore what a learner is and how it fits into the mlr3 package.

17.2.2.1. What is a "Learner"?

Think of a learner as the method or tool that the computer uses to learn from the data. Another common name for a "learner" is a "model." It's similar to choosing a tutor or a teacher for a student. Different learners have different ways of understanding and processing information. For example:

- Some learners might be great at recognizing patterns in data, like a tutor who is excellent at spotting trends.
- Others might be good at making decisions based on rules, like a tutor who uses step-by-step logic.

In machine learning, there are many types of learners, each with its own strengths and weaknesses. Here are a few examples:

- **Decision Trees:** These learners make decisions by asking a series of questions, like "Is the house larger than 1000 square feet?" and "Does it have more than 3 bedrooms?"
- **k-Nearest Neighbors:** These learners make predictions based on the similarity of new data points to existing data points.
- **Linear Regression:** This learner tries to fit a straight line through the data points to make predictions about numbers.
- **Random Forests:** These are like a group of decision trees working together to make more accurate predictions.

- **Support Vector Machines:** These learners find the best boundary that separates different categories in the data.

17.2.2.2. Choosing the Right Learner

Selecting the right learner is crucial because different learners work better for different types of tasks and data. For example:

- If your task is to classify emails as spam or not spam, a decision tree or a support vector machine might work well.
- If you're predicting house prices, linear regression or random forests could be good choices.

The goal is to find a learner that can understand the patterns in your data and make accurate predictions. This is where the mlr3 package comes in handy. It provides a wide range of learners that you can choose from, making it easier to experiment and find the best learner for your task.

17.2.2.3. Learners in mlr3

In the mlr3 package, learners are pre-built tools that you can easily use for your tasks. Here's how it works:

1. **Select a Learner:** mlr3 provides a variety of learners to choose from, like decision trees, linear regression, and more.
2. **Train the Learner:** Once you've selected a learner, you provide it with your task (the problem, data, and target). The learner uses this information to learn and make predictions.
3. **Evaluate and Improve:** After training, you can test how well the learner performs and make adjustments if needed, such as trying a different learner or fine-tuning the current one.

17.2.2.4. mlr3 and Learners

Objects of class `Learner` provide a unified interface to many popular machine learning algorithms in R. They consist of methods to train and predict a model for a `Task` and provide meta-information about the learners, such as the hyperparameters (which control the behavior of the learner) you can set.

The base class of each learner is `Learner`, specialized for regression as `LearnerRegr` and for classification as `LearnerClassif`. Other types of learners, provided by extension packages, also inherit from the `Learner` base class, e.g. `mlr3proba::LearnerSurv` or `mlr3cluster::LearnerClust`.

All Learners work in a two-stage procedure:

- **Training stage**: The training data (features and target) is passed to the Learner's `$train()` function which trains and stores a model, i.e. the relationship of the target and features.
- **Predict stage**: The new data, usually a different slice of the original data than used for training, is passed to the `$predict()` method of the Learner. The model trained in the first step is used to predict the missing target, e.g. labels for classification problems or the numerical value for regression problems.

There are a number of predefined learners. The mlr3 package ships with the following set of classification and regression learners. We deliberately keep this small to avoid unnecessary dependencies:

- `classif.featureless`: Simple baseline classification learner. The default is to always predict the label that is most frequent in the training set. While this is not very useful by itself, it can be used as a "fallback learner" to make predictions in case another, more sophisticated, learner failed for some reason.
- `regr.featureless`: Simple baseline regression learner. The default is to always predict the mean of the target in training set. Similar to `mlr_learners_classif.featureless`, it makes for a good "fallback learner"
- `classif.rpart`: Single classification tree from package rpart.
- `regr.rpart`: Single regression tree from package rpart.

This set of baseline learners is usually insufficient for a real data analysis. Thus, we have cherry-picked implementations of the most popular machine learning method and collected them in the mlr3learners package:

- Linear (`regr.lm`) and logistic (`classif.log_reg`) regression
- Penalized Generalized Linear Models (`regr.glmnet`, `classif.glmnet`), possibly with built-in optimization of the penalization parameter (`regr.cv_glmnet`, `classif.cv_glmnet`)
- (Kernelized) k-Nearest Neighbors regression (`regr.kknn`) and classification (`classif.kknn`).
- Kriging / Gaussian Process Regression (`regr.km`)
- Linear (`classif.lda`) and Quadratic (`classif.qda`) Discriminant Analysis
- Naive Bayes Classification (`classif.naive_bayes`)
- Support-Vector machines (`regr.svm`, `classif.svm`)

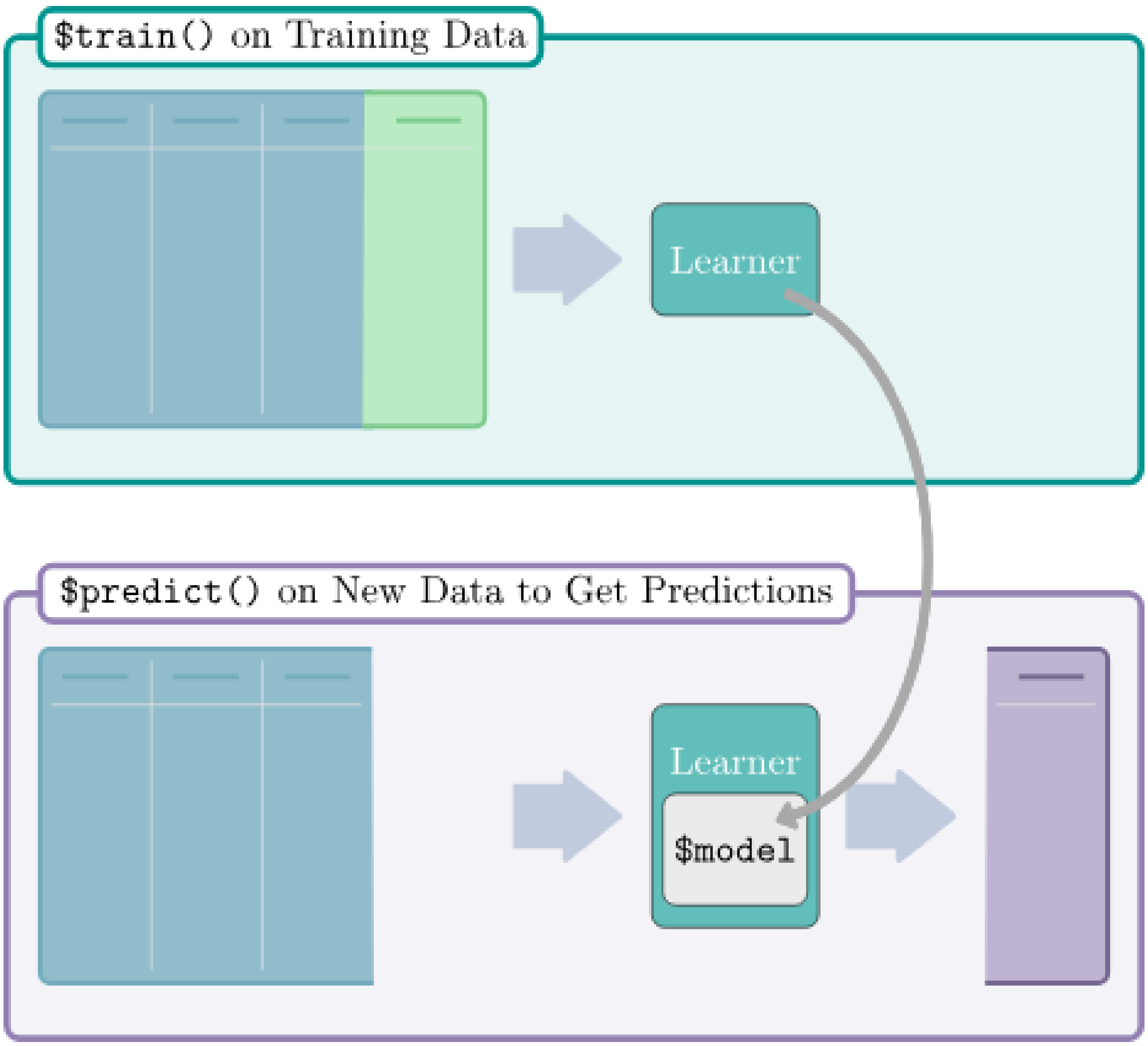


Figure 17.3.: Two stages of a learner. Top: data (features and a target) are passed to an (untrained) learner. Bottom: new data are passed to the trained model which makes predictions for the 'missing' target column.

- Gradient Boosting (`regr.xgboost`, `classif.xgboost`)
- Random Forests for regression and classification (`regr.ranger`, `classif.ranger`)

More machine learning methods and alternative implementations are collected in the mlr3extralearners repository.

17.3. Setup

```
library(mlr3verse)
library(GEOquery)
library(mlr3learners) # for knn
library(ranger) # for randomforest
set.seed(789)
```

17.4. Example: Cancer types

In this exercise, we will be classifying cancer types based on gene expression data. The data we are going to access are from Brouwer-Visser et al. (2018).

The data are from the Gene Expression Omnibus (GEO) database, a public repository of functional genomics data. The data are from a study that aimed to identify gene expression signatures that can distinguish between different types of cancer. The data include gene expression profiles from patients with different types of cancer. The goal is to build a machine learning model that can predict the cancer type based on the gene expression data.

17.4.1. Understanding the Problem

Before we start building a machine learning model, it's important to understand the problem we are trying to solve. Here are some key questions to consider:

- What are the features?
- What is the target variable?
- What type of machine learning task is this (classification, regression, clustering)?
- What is the goal of the analysis?

17.4.2. Data Preparation

Use the GEOquery package to fetch data about GSE103512.

```
library(GEOquery)
gse = getGEO("GSE103512")[[1]]
```

The first step, a detail, is to convert from the older Bioconductor data structure (GEOquery was written in 2007), the `ExpressionSet`, to the newer `SummarizedExperiment`.

```
library(SummarizedExperiment)
se = as(gse, "SummarizedExperiment")
```

Examine two variables of interest, cancer type and tumor/normal status.

```
with(colData(se),table(`cancer.type.ch1`,`normal.ch1`))
```

```
               normal.ch1
cancer.type.ch1 no yes
          BC    65  10
          CRC   57  12
          NSCLC 60   9
          PCA   60   7
```

Before embarking on a machine learning analysis, we need to make sure that we understand the data. Things like missing values, outliers, and other problems can cause problems for machine learning algorithms.

In R, plotting, summaries, and other exploratory data analysis tools are available. PCA analysis, clustering, and other methods can also be used to understand the data. It is worth spending time on this step, as it can save time later.

17.4.3. Feature selection and data cleaning

While we could use all genes in the analysis, we will select the most informative genes using the variance of gene expression across samples. Other methods for feature selection are available, including those based on correlation with the outcome variable.

> **!** Feature selection
>
> Feature selection should be done on the training data only, not the test data to avoid overfitting. The test data should be used only for evaluation. In other words, the test data should be "unseen" by the model until the final evaluation.

Remember that the `apply` function applies a function to each row or column of a matrix. Here, we apply the `sd` function to each row of the expression matrix to get a vector of stan

```
sds = apply(assay(se, 'exprs'),1,sd)
## filter out normal tissues
se_small = se[order(sds,decreasing = TRUE)[1:200],
              colData(se)$characteristics_ch1.1=='normal: no']
# remove genes with no gene symbol
se_small = se_small[rowData(se_small)$Gene.Symbol!='',]
```

To make the data easier to work with, we will use the opportunity to use one of the rowData columns as the rownames of the data frame. The `make.names` function is used to make sure that the rownames are valid R variable names and unique.

```
## convert to matrix for later use
dat = assay(se_small, 'exprs')
rownames(dat) = make.names(rowData(se_small)$Gene.Symbol)
```

We also need to transpose the data so that the rows are the samples and the columns are the features in order to use the data with mlr3.

```
feat_dat = t(dat)
tumor = data.frame(tumor_type = colData(se_small)$cancer.type.ch1, feat_dat)
```

This is another good time to check the data. Make sure that the data is in the format that you expect. Check the dimensions, the column names, and the data types.

17.4.4. Creating the "task"

The first step in using mlr3 is to create a task. A task is a data set with a target variable. In this case, the target variable is the cancer type. The mlr3 package provides a function to convert a data frame into a task. These tasks can be used with any machine learning algorithm in mlr3.

This is a classification task, so we will use the `as_task_classif` function to create the task. The classification task requires a target variable that is categorical.

```
library(mlr3)
tumor$tumor_type = as.factor(tumor$tumor_type)
task = as_task_classif(tumor,target='tumor_type')
```

17.4.5. Splitting the data

Here, we randomly divide the data into 2/3 training data and 1/3 test data. This is a common split, but other splits can be used. The training data is used to train the model, and the test data is used to evaluate the trained model.

```
set.seed(7)
train_set = sample(task$row_ids, 0.67 * task$nrow)
test_set = setdiff(task$row_ids, train_set)
```

> **!** Important
>
> Training and testing on the same data is a common mistake. We want to test the model on data that it has not seen before. This is the only way to know if the model is overfitting and to get an accurate estimate of the model's performance.

In the next sections, we will train and evaluate three different models on the data: k-nearest-neighbor, classification tree, and random forest. Remember that the goal is to predict the cancer type based on the gene expression data. The mlr3 package uses the concept of "learners" to encapsulate different machine learning algorithms.

17.4.6. Example learners

17.4.6.1. K-nearest-neighbor

The first model we will use is the k-nearest-neighbor model. This model is based on the idea that similar samples have similar outcomes. The number of neighbors to use is a parameter that can be tuned. We'll use the default value of 7, but you can try other values to see how they affect the results. In fact, mlr3 provides the ability to tune parameters automatically, but we won't cover that here.

17.4.6.1.1. Create the learner

In mlr3, the machine learning algorithms are called learners. To create a learner, we use the `lrn` function. The `lrn` function takes the name of the learner as an argument. The `lrn` function also takes other arguments that are specific to the learner. In this case, we will use the default values for the arguments.

```
library(mlr3learners)
learner = lrn("classif.kknn")
```

You can get a list of all the learners available in mlr3 by using the `lrn()` function without any arguments.

```
lrn()
```

```
<DictionaryLearner> with 49 stored values
Keys: classif.cv_glmnet, classif.debug, classif.featureless,
  classif.glmnet, classif.kknn, classif.lda, classif.log_reg,
  classif.multinom, classif.naive_bayes, classif.nnet, classif.qda,
  classif.ranger, classif.rpart, classif.svm, classif.xgboost,
  clust.agnes, clust.ap, clust.cmeans, clust.cobweb, clust.dbscan,
  clust.dbscan_fpc, clust.diana, clust.em, clust.fanny,
  clust.featureless, clust.ff, clust.hclust, clust.hdbscan,
  clust.kkmeans, clust.kmeans, clust.MBatchKMeans, clust.mclust,
  clust.meanshift, clust.optics, clust.pam, clust.SimpleKMeans,
  clust.xmeans, regr.cv_glmnet, regr.debug, regr.featureless,
  regr.glmnet, regr.kknn, regr.km, regr.lm, regr.nnet, regr.ranger,
  regr.rpart, regr.svm, regr.xgboost
```

17.4.6.1.2. Train

To train the model, we use the `train` function. The `train` function takes the task and the row ids of the training data as arguments.

```
learner$train(task, row_ids = train_set)
```

Here, we can look at the trained model:

```
# output is large, so do this on your own
learner$model
```

17.4.6.1.3. Predict

Lets use our trained model works to predict the classes of the **training** data. Of course, we already know the classes of the training data, but this is a good way to check that the model is working as expected. It also gives us a measure of performance on the training data that we can compare to the test data to look for overfitting.

```
pred_train = learner$predict(task, row_ids=train_set)
```

And check on the test data:

```
pred_test = learner$predict(task, row_ids=test_set)
```

17.4.6.1.4. Assess

In this section, we can look at the accuracy and performance of our model on the training data and the test data. We can also look at the confusion matrix to see which classes are being confused with each other.

```
pred_train$confusion
```

```
        truth
response BC CRC NSCLC PCA
   BC    42   0     0   0
   CRC    0  40     0   0
   NSCLC  1   0    44   0
   PCA    0   0     0  35
```

This is a multi-class confusion matrix. The rows are the true classes and the columns are the predicted classes. The diagonal shows the number of samples that were correctly classified. The off-diagonal elements show the number of samples that were misclassified.

We can also look at the accuracy of the model on the training data and the test data. The accuracy is the number of correctly classified samples divided by the total number of samples.

```
measures = msrs(c('classif.acc'))
pred_train$score(measures)
```

```
classif.acc
  0.9938272
```

```
pred_test$confusion
```

```
        truth
response BC CRC NSCLC PCA
   BC    22   0     0   0
   CRC    0  17     1   0
   NSCLC  0   0    15   0
   PCA    0   0     0  25
```

```
pred_test$score(measures)
```

```
classif.acc
     0.9875
```

Compare the accuracy on the training data to the accuracy on the test data. Do you see any evidence of overfitting?

17.4.6.2. Classification tree

We are going to use a classification tree to classify the data. A classification tree is a series of yes/no questions that are used to classify the data. The questions are based on the features in the data. The classification tree is built by finding the feature that best separates the data into the different classes. Then, the data is split based on the value of that feature. The process is repeated until the data is completely separated into the different classes.

17.4.6.2.1. Train

```
# in this case, we want to keep the model
# so we can look at it later
learner = lrn("classif.rpart", keep_model = TRUE)
```

```
learner$train(task, row_ids = train_set)
```

We can take a look at the model.

```
learner$model
```

```
n= 162

node), split, n, loss, yval, (yprob)
      * denotes terminal node

 1) root 162 118 NSCLC (0.26543210 0.24691358 0.27160494 0.21604938)
   2) CDHR5>=5.101625 40   0 CRC (0.00000000 1.00000000 0.00000000 0.00000000) *
   3) CDHR5< 5.101625 122  78 NSCLC (0.35245902 0.00000000 0.36065574 0.28688525)
     6) ACPP< 6.088431 87  43 NSCLC (0.49425287 0.00000000 0.50574713 0.00000000)
      12) GATA3>=4.697803 41   1 BC (0.97560976 0.00000000 0.02439024 0.00000000) *
      13) GATA3< 4.697803 46   3 NSCLC (0.06521739 0.00000000 0.93478261 0.00000000) *
     7) ACPP>=6.088431 35   0 PCA (0.00000000 0.00000000 0.00000000 1.00000000) *
```

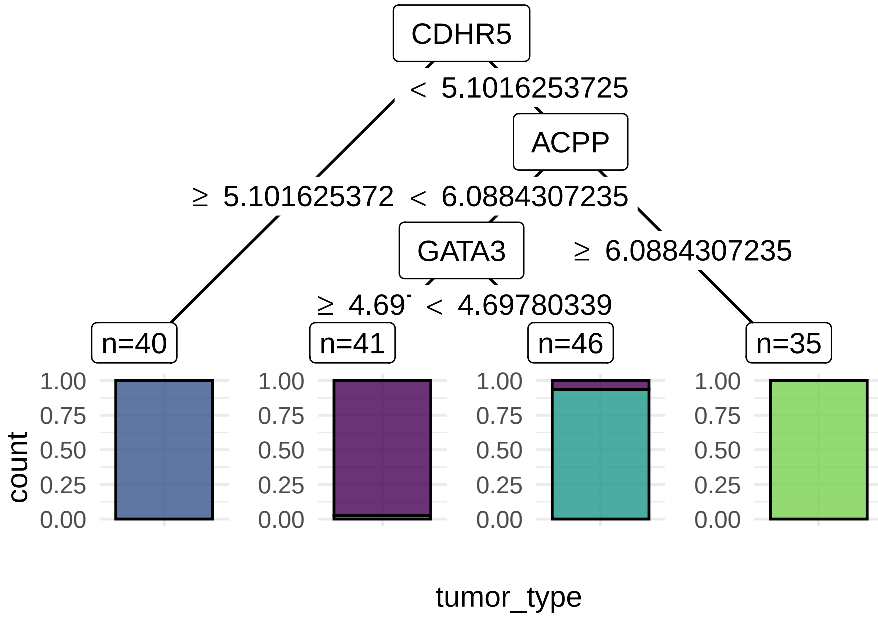
Decision trees are easy to visualize if they are small. Here, we can see that the tree is very simple, with only two splits.

```
library(mlr3viz)
library(ggparty)
```

```
Loading required package: ggplot2

Loading required package: partykit

Loading required package: grid

Loading required package: libcoin
```

```
Loading required package: mvtnorm


Attaching package: 'partykit'

The following object is masked from 'package:SummarizedExperiment':

    width

The following object is masked from 'package:GenomicRanges':

    width

The following object is masked from 'package:IRanges':

    width

The following object is masked from 'package:S4Vectors':

    width

The following object is masked from 'package:BiocGenerics':

    width
```

```r
autoplot(learner, type='ggparty')
```



17.4.6.2.2. Predict

Now that we have trained the model on the *training* data, we can use it to predict the classes of the training data and the test data. The `$predict` method takes a `task` and produces a prediction based on the *trained* model, in this case, called `learner`.

```
pred_train = learner$predict(task, row_ids=train_set)
```

Remember that we split the data into training and test sets. We can use the trained model to predict the classes of the test data. Since the *test* data was not used to train the model, it is not "cheating" like what we just did where we did the prediction on the *training* data.

```
pred_test = learner$predict(task, row_ids=test_set)
```

17.4.6.2.3. Assess

For classification tasks, we often look at a confusion matrix of the *truth* vs the *predicted* classes for the samples.

> **!** Important
>
> Assessing the performance of a model should **always** be **reported** from assessment on an independent test set.

```
pred_train$confusion
```

```
        truth
response BC CRC NSCLC PCA
   BC    40   0     1   0
   CRC    0  40     0   0
   NSCLC  3   0    43   0
   PCA    0   0     0  35
```

- What does this confusion matrix tell you?

We can also ask for several "measures" of the performance of the model. Here, we ask for the accuracy of the model. To get a complete list of measures, use `msr()`.

```
measures = msrs(c('classif.acc'))
pred_train$score(measures)
```

```
classif.acc
  0.9753086
```

- How does the accuracy compare to the confusion matrix?
- How does this accuracy compare to the accuracy of the k-nearest-neighbor model?
- How about the decision tree model?

```
pred_test$confusion
```

```
        truth
response BC CRC NSCLC PCA
   BC    20   0     1   0
   CRC    0  17     3   0
   NSCLC  2   0    12   0
   PCA    0   0     0  25
```

```
pred_test$score(measures)
```

```
classif.acc
      0.925
```

- What does the confusion matrix in the *test* set tell you?
- How do the assessments of the *test* and *training* sets differ?

> Overfitting
>
> When the assessment of the test set is worse than the evaluation of the training set, the model may be *overfit*. How to address overfitting varies by model type, but it is a sign that you should pay attention to model selection and parameters.

17.4.6.3. RandomForest

```
learner = lrn("classif.ranger", importance = "impurity")
```

17.4.6.3.1. Train

```
learner$train(task, row_ids = train_set)
```

Again, you can look at the model that was trained.

```
learner$model
```

```
Ranger result

Call:
 ranger::ranger(dependent.variable.name = task$target_names, data = task$data(),      proba

Type:                             Classification
Number of trees:                  500
Sample size:                      162
Number of independent variables:  192
Mtry:                             13
```

```
Target node size:                 1
Variable importance mode:         impurity
Splitrule:                        gini
OOB prediction error:             0.62 %
```

For more details, the mlr3 random forest approach is based ont he ranger package. You can look at the ranger documentation.

- What is the OOB error in the output?

Random forests are a collection of decision trees. Since predictors enter the trees in a random order, the trees are different from each other. The random forest procedure gives us a measure of the "importance" of each variable.

```r
head(learner$importance(), 15)
```

```
   CDHR5  TRPS1.1    FABP1   EPS8L3    KRT20    EFHD1   LGALS4    TRPS1 
4.791870 3.918063 3.692649 3.651422 3.340382 3.314491 2.952969 2.926175 
   SFTPB  SFTPB.1    GATA3  GATA3.1  TMPRSS2    MUC12    POF1B 
2.805811 2.681004 2.344603 2.271845 2.248734 2.207347 1.806906 
```

More "important" variables are those that are more often used in the trees. Are the most important variables the same as the ones that were important in the decision tree?

If you are interested, look up a few of the important variables in the model to see if they make biological sense.

17.4.6.3.2. Predict

Again, we can use the trained model to predict the classes of the training data and the test data.

```r
pred_train = learner$predict(task, row_ids=train_set)
```

```r
pred_test = learner$predict(task, row_ids=test_set)
```

17.4.6.3.3. Assess

```r
pred_train$confusion
```

```
        truth
response BC CRC NSCLC PCA
   BC    43   0     0   0
   CRC    0  40     0   0
   NSCLC  0   0    44   0
   PCA    0   0     0  35
```

```r
measures = msrs(c('classif.acc'))
pred_train$score(measures)
```

```
classif.acc
          1
```

```r
pred_test$confusion
```

```
        truth
response BC CRC NSCLC PCA
   BC    22   0     0   0
   CRC    0  17     0   0
   NSCLC  0   0    16   0
   PCA    0   0     0  25
```

```r
pred_test$score(measures)
```

```
classif.acc
          1
```

17.5. Example Predicting age from DNA methylation

We will be building a regression model for chronological age prediction, based on DNA methylation. This is based on the work of Jana Naue et al. 2017, in which biomarkers are examined to predict the chronological age of humans by analyzing the DNA methylation patterns. Different machine learning algorithms are used in this study to make an age prediction.

It has been recognized that within each individual, the level of DNA methylation changes with age. This knowledge is used to select useful biomarkers from DNA methylation datasets. The CpG sites with the highest correlation to age are selected as the biomarkers (and therefore features for building a regression model). In this tutorial, specific biomarkers are analyzed by machine learning algorithms to create an age prediction model.

The data are taken from this tutorial.

```
library(data.table)
meth_age = rbind(
    fread('https://zenodo.org/record/2545213/files/test_rows_labels.csv'),
    fread('https://zenodo.org/record/2545213/files/train_rows.csv')
)
```

Let's take a quick look at the data.

```
head(meth_age)
```

```
   RPA2_3 ZYG11A_4  F5_2 HOXC4_1 NKIRAS2_2 MEIS1_1 SAMD10_2 GRM2_9 TRIM59_5
    <num>    <num> <num>   <num>     <num>   <num>    <num>  <num>    <num>
1:  65.96    18.08 41.57   55.46     30.69   63.42    40.86  68.88    44.32
2:  66.83    20.27 40.55   49.67     29.53   30.47    37.73  53.30    50.09
3:  50.30    11.74 40.17   33.85     23.39   58.83    38.84  35.08    35.90
4:  65.54    15.56 33.56   36.79     20.23   56.39    41.75  50.37    41.46
5:  59.01    14.38 41.95   30.30     24.99   54.40    37.38  30.35    31.28
6:  81.30    14.68 35.91   50.20     26.57   32.37    32.30  55.19    42.21
   LDB2_3 ELOVL2_6 DDO_1 KLF14_2   Age
    <num>    <num> <num>   <num> <int>
1:  56.17    62.29 40.99    2.30    40
2:  58.40    61.10 49.73    1.07    44
3:  58.81    50.38 63.03    0.95    28
4:  58.05    50.58 62.13    1.99    37
5:  65.80    48.74 41.88    0.90    24
6:  70.15    61.36 33.62    1.87    43
```

As before, we create the `task` object, but this time we use `as_task_regr()` to create a regression task.

- Why is this a regression task?

```
task = as_task_regr(meth_age,target = 'Age')
```

```
set.seed(7)
train_set = sample(task$row_ids, 0.67 * task$nrow)
test_set = setdiff(task$row_ids, train_set)
```

17.5.1. Example learners

17.5.1.1. Linear regression

We will start with a simple linear regression model.

```
learner = lrn("regr.lm")
```

17.5.1.1.1. Train

```
learner$train(task, row_ids = train_set)
```

When you train a linear regression model, we can evaluate some of the diagnostic plots to see if the model is appropriate (Figure 17.4).

```
par(mfrow=c(2,2))
plot(learner$model)
```

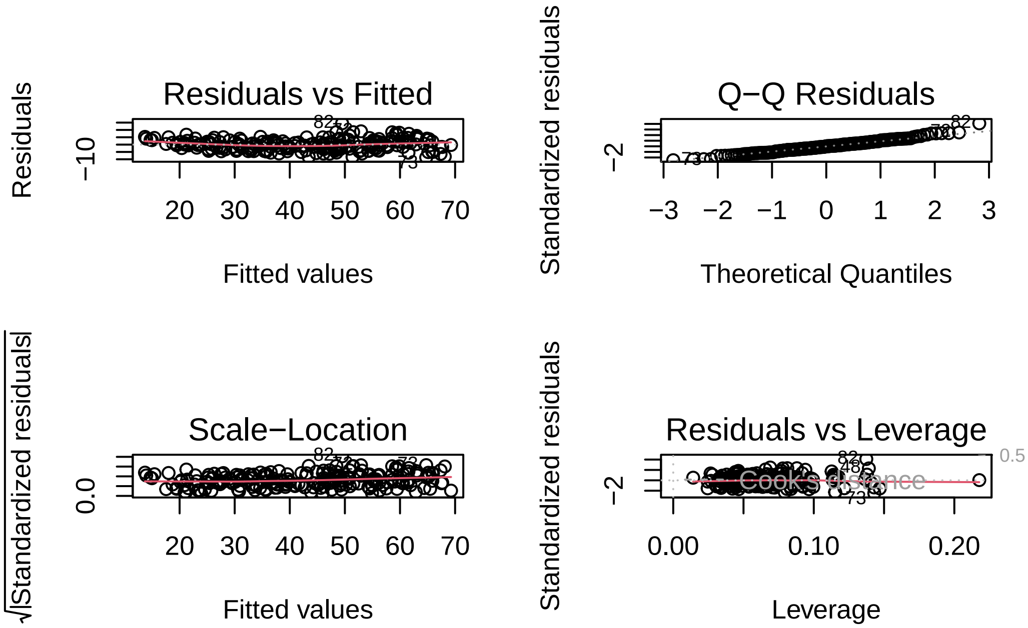


Figure 17.4.: Regression diagnostic plots. The top left plot shows the residuals vs. fitted values. The top right plot shows the normal Q-Q plot. The bottom left plot shows the scale-location plot. The bottom right plot shows the residuals vs. leverage.

17.5.1.1.2. Predict

```
pred_train = learner$predict(task, row_ids=train_set)
```

```
pred_test = learner$predict(task, row_ids=test_set)
```

17.5.1.1.3. Assess

```
pred_train
```

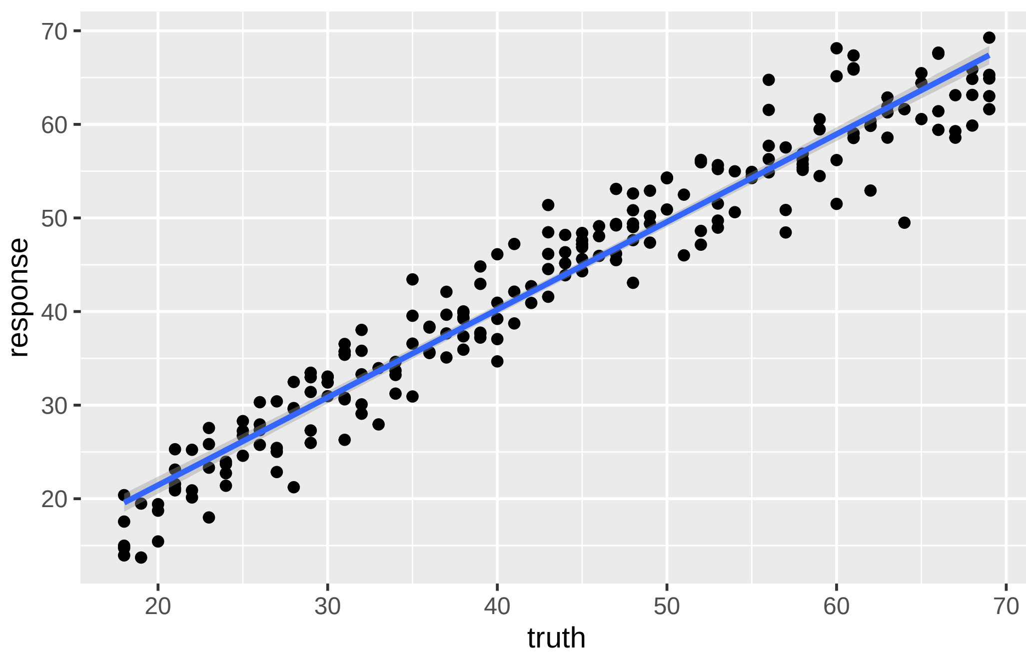
```
<PredictionRegr> for 209 observations:
    row_ids truth response
        298    29 31.40565
        103    58 56.26019
        194    53 48.96480
```

```
---
        312     48 52.61195
        246     66 67.66312
        238     38 39.38414
```

We can plot the relationship between the **truth** and **response**, or predicted value to see visually how our model performs.

```
library(ggplot2)
ggplot(pred_train,aes(x=truth, y=response)) +
    geom_point() +
    geom_smooth(method='lm')
```

```
`geom_smooth()` using formula = 'y ~ x'
```



We can use the r-squared of the fit to roughly compare two models.

```
measures = msrs(c('regr.rsq'))
pred_train$score(measures)
```

```
 regr.rsq
0.9376672
```

```r
pred_test
```

```
<PredictionRegr> for 103 observations:
    row_ids truth response
          4    37 37.64301
          5    24 28.34777
          7    34 33.22419
---
        306    42 41.65864
        307    63 58.68486
        309    68 70.41987
```

```r
pred_test$score(measures)
```

```
 regr.rsq
0.9363526
```

17.5.1.2. Regression tree

```r
learner = lrn("regr.rpart", keep_model = TRUE)
```

17.5.1.2.1. Train

```r
learner$train(task, row_ids = train_set)
```

```r
learner$model
```

```
n= 209

node), split, n, deviance, yval
      * denotes terminal node

 1) root 209 45441.4500 43.27273
```

```
   2) ELOVL2_6< 56.675 98  5512.1220 30.24490
     4) ELOVL2_6< 47.24 47   866.4255 24.23404
       8) GRM2_9< 31.3 34   289.0588 22.29412 *
       9) GRM2_9>=31.3 13   114.7692 29.30769 *
     5) ELOVL2_6>=47.24 51  1382.6270 35.78431
      10) F5_2>=39.295 35   473.1429 33.28571 *
      11) F5_2< 39.295 16   213.0000 41.25000 *
   3) ELOVL2_6>=56.675 111  8611.3690 54.77477
     6) ELOVL2_6< 65.365 63  3101.2700 49.41270
      12) KLF14_2< 3.415 37  1059.0270 46.16216 *
      13) KLF14_2>=3.415 26  1094.9620 54.03846 *
     7) ELOVL2_6>=65.365 48  1321.3120 61.81250 *
```

What is odd about using a regression tree here is that we end up with only a few discrete estimates of age. Each "leaf" has a value.

17.5.1.2.2. Predict

```
pred_train = learner$predict(task, row_ids=train_set)
```

```
pred_test = learner$predict(task, row_ids=test_set)
```

17.5.1.2.3. Assess

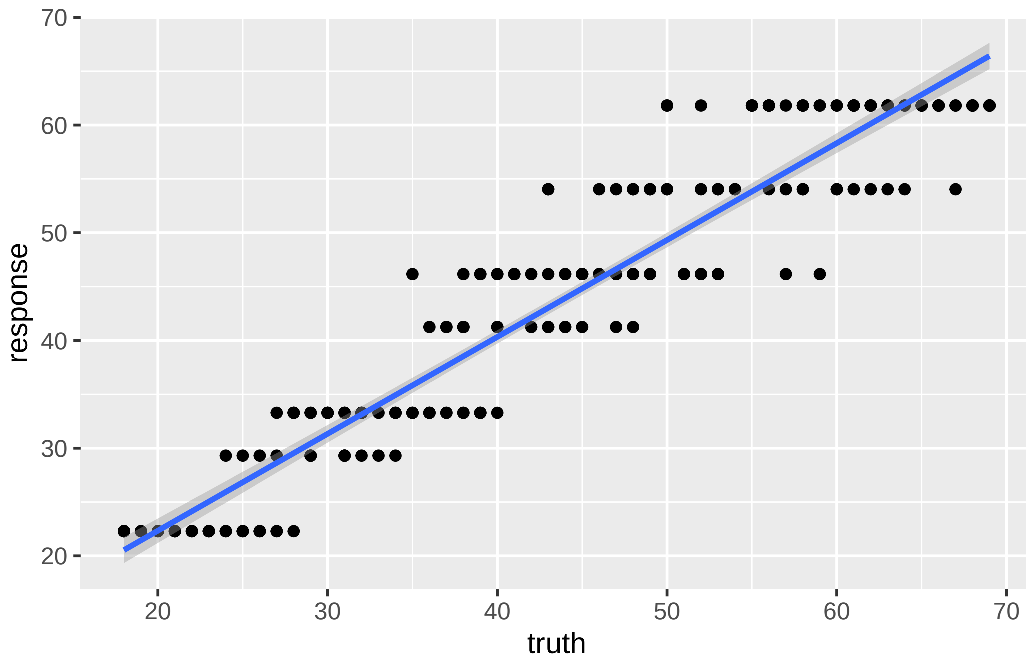
```
pred_train
```

```
<PredictionRegr> for 209 observations:
    row_ids truth response
        298    29 33.28571
        103    58 61.81250
        194    53 46.16216
---
        312    48 54.03846
        246    66 61.81250
        238    38 41.25000
```

We can see the effect of the discrete values much more clearly here.

```
library(ggplot2)
ggplot(pred_train,aes(x=truth, y=response)) +
    geom_point() +
    geom_smooth(method='lm')
```

```
`geom_smooth()` using formula = 'y ~ x'
```



And the r-squared values for this model prediction shows quite a bit of difference from the linear regression above.

```
measures = msrs(c('regr.rsq'))
pred_train$score(measures)
```

```
 regr.rsq
0.8995351
```

```
pred_test
```

```
<PredictionRegr> for 103 observations:
    row_ids truth response
          4    37 41.25000
          5    24 33.28571
          7    34 33.28571
---
        306    42 46.16216
        307    63 61.81250
        309    68 61.81250
```

```r
pred_test$score(measures)
```

```
 regr.rsq
0.8545402
```

17.5.1.3. RandomForest

Randomforest is also tree-based, but unlike the single regression tree above, randomforest is a "forest" of trees which will eliminate the discrete nature of a single tree.

```r
learner = lrn("regr.ranger", mtry=2, min.node.size=20)
```

17.5.1.3.1. Train

```r
learner$train(task, row_ids = train_set)
```

```r
learner$model
```

```
Ranger result

Call:
 ranger::ranger(dependent.variable.name = task$target_names, data = task$data(),      case.

Type:                             Regression
Number of trees:                  500
Sample size:                      209
Number of independent variables:  13
```

```
Mtry:                             2
Target node size:                 20
Variable importance mode:         none
Splitrule:                        variance
OOB prediction error (MSE):       18.85364
R squared (OOB):                  0.9137009
```

17.5.1.3.2. Predict

```r
pred_train = learner$predict(task, row_ids=train_set)
```

```r
pred_test = learner$predict(task, row_ids=test_set)
```

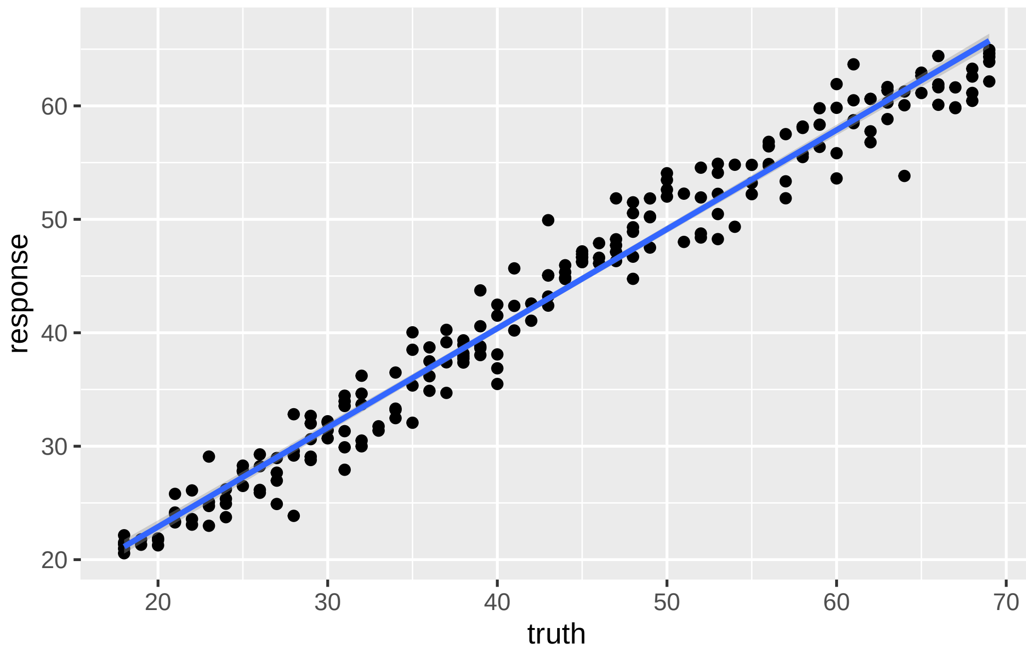
17.5.1.3.3. Assess

```r
pred_train
```

```
<PredictionRegr> for 209 observations:
    row_ids truth response
        298    29 30.62154
        103    58 58.05445
        194    53 48.25661
---
        312    48 51.49846
        246    66 64.39315
        238    38 38.18038
```

```r
ggplot(pred_train,aes(x=truth, y=response)) +
    geom_point() +
    geom_smooth(method='lm')
```

```
`geom_smooth()` using formula = 'y ~ x'
```




```
measures = msrs(c('regr.rsq'))
pred_train$score(measures)
```

```
regr.rsq
0.960961
```

```
pred_test
```

```
<PredictionRegr> for 103 observations:
    row_ids truth response
          4    37 37.79631
          5    24 29.18371
          7    34 33.26780
---
        306    42 40.29101
        307    63 58.26534
        309    68 63.15481
```

```
pred_test$score(measures)
```

```
 regr.rsq
0.9208394
```

17.6. Example: Expression prediction from histone modification data

In this little set of exercises, you will be using histone marks near a gene to predict its expression (Figure 17.5).

$$y\ h1 + h2 + h3 + \ldots \tag{17.1}$$

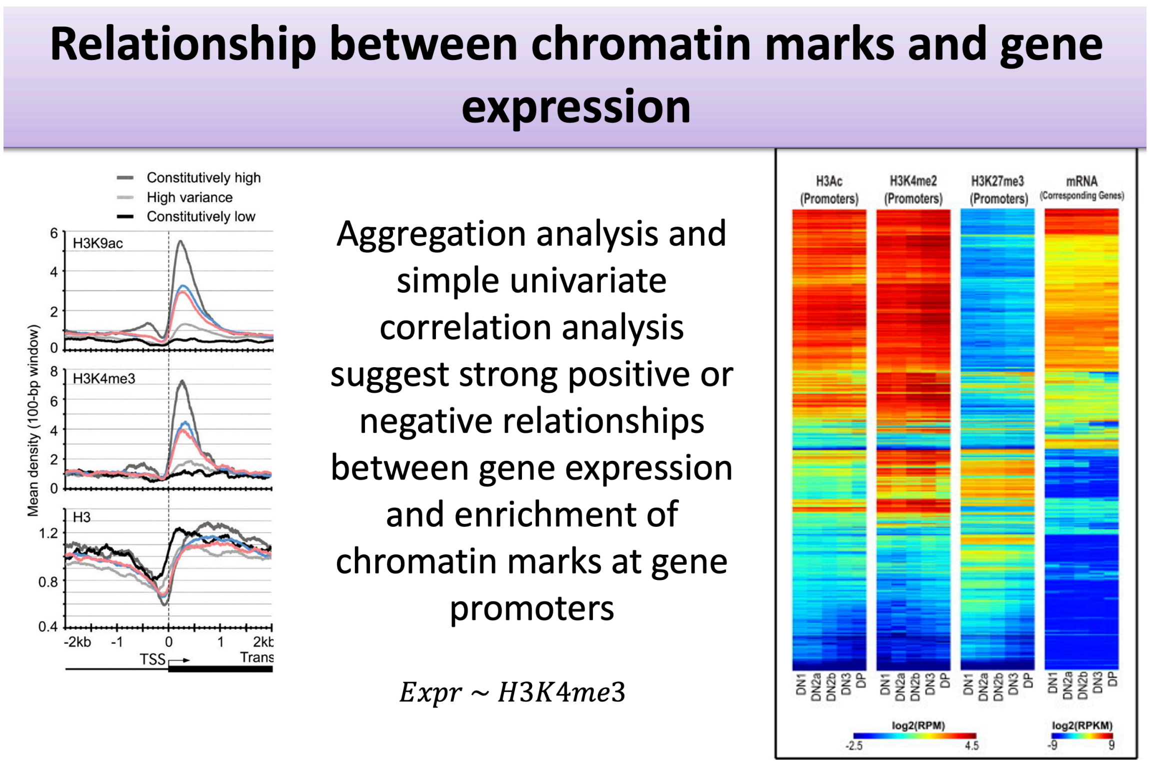


Figure 17.5.: What is the combined effect of histone marks on gene expression?

The data are from a study that aimed to predict gene expression from histone modification data. The data include gene expression levels and histone modification data for a set of genes. The goal is to build a machine learning model that can predict gene expression levels based on the histone modification data. The histone modification data are simply summaries of the histone marks within the promoter, defined as the region 2kb upstream of the transcription start site for this exercise.

We will try a couple of different approaches:

1. Penalized regression
2. RandomForest

17.6.1. The Data

The data in this exercise were developed by Anshul Kundaje. We are not going to focus on the details of the data collection, etc. Instead, this is

```
fullFeatureSet <- read.table("http://seandavi.github.io/ITR/expression-prediction/features.
```

What are the column names of the predictor variables?

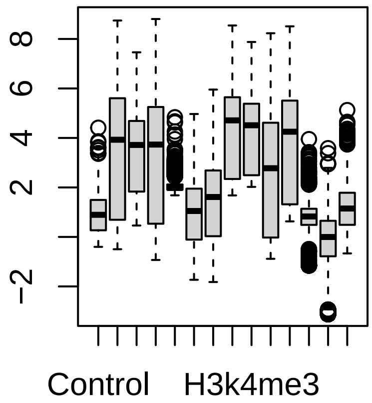
```
colnames(fullFeatureSet)
```

```
 [1] "Control"  "Dnase"    "H2az"     "H3k27ac"  "H3k27me3" "H3k36me3"
 [7] "H3k4me1"  "H3k4me2"  "H3k4me3"  "H3k79me2" "H3k9ac"   "H3k9me1"
[13] "H3k9me3"  "H4k20me1"
```

These are going to be predictors combined into a model. Some of our learners will rely on predictors being on a similar scale. Are our data already there?

To perform centering and scaling by column, we can convert to a matrix and then use `scale`.

```
par(mfrow=c(1,2))
scaled_features <- scale(as.matrix(fullFeatureSet))
boxplot(fullFeatureSet, title='Original data')
boxplot(scaled_features, title='Centered and scaled data')
```



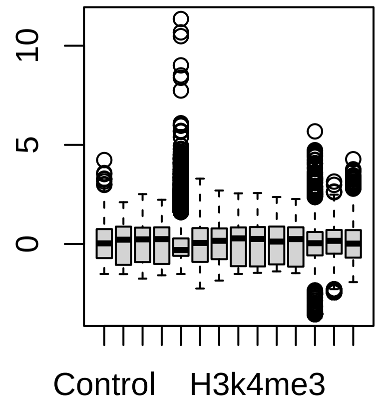


Figure 17.6.: Boxplots of original and scaled data.

There is a row for each gene and a column for each histone mark and we can see that the data are centered and scaled by column. We can also see some patterns in the data (see Figure 17.7).

```
sampled_features <- fullFeatureSet[sample(nrow(scaled_features), 500),]
library(ComplexHeatmap)
```

```
========================================
ComplexHeatmap version 2.20.0
Bioconductor page: http://bioconductor.org/packages/ComplexHeatmap/
Github page: https://github.com/jokergoo/ComplexHeatmap
Documentation: http://jokergoo.github.io/ComplexHeatmap-reference

If you use it in published research, please cite either one:
- Gu, Z. Complex Heatmap Visualization. iMeta 2022.
- Gu, Z. Complex heatmaps reveal patterns and correlations in multidimensional
    genomic data. Bioinformatics 2016.


The new InteractiveComplexHeatmap package can directly export static
complex heatmaps into an interactive Shiny app with zero effort. Have a try!

This message can be suppressed by:
  suppressPackageStartupMessages(library(ComplexHeatmap))
========================================
```

```
Heatmap(sampled_features, name='histone marks', show_row_names=FALSE)
```

```
Warning: The input is a data frame-like object, convert it to a matrix.
```

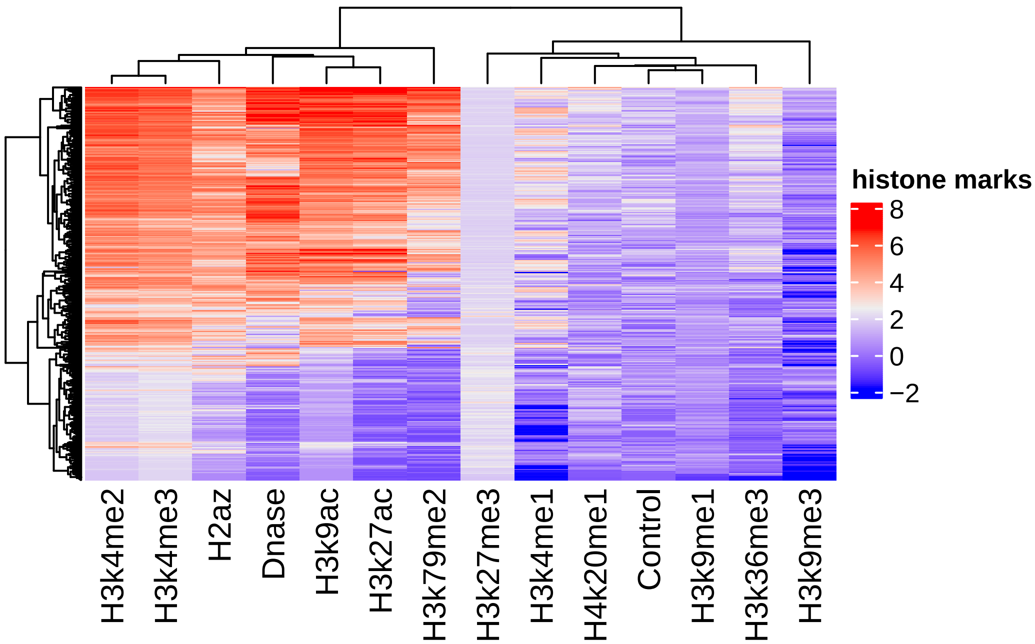


Figure 17.7.: Heatmap of 500 randomly sampled rows of the data. Columns are histone marks and there is a row for each gene.

Now, we can read in the associated gene expression measures that will become our "target" for prediction.

```
target <- scan(url("http://seandavi.github.io/ITR/expression-prediction/target.txt"), skip=
# make into a dataframe
exp_pred_data <- data.frame(gene_expression=target, scaled_features)
```

And the first few rows of the target data frame using:

```
head(exp_pred_data,3)
```

```
                              gene_expression    Control      Dnase       H2az
ENSG00000000419.7.49575069          6.082343  0.7452926  0.7575546  1.0728432
ENSG00000000457.8.169863093         2.989145  1.9509786  1.0216546  0.3702787
ENSG00000000938.7.27961645         -5.058894 -0.3505542 -1.4482958 -1.0390775
                               H3k27ac   H3k27me3   H3k36me3    H3k4me1
ENSG00000000419.7.49575069   1.0950440 -0.5125312  1.1334793  0.4127984
ENSG00000000457.8.169863093  0.7142157 -0.4079244  0.8739005  1.1649282
ENSG00000000938.7.27961645  -1.0173283  1.4117293 -0.5157582 -0.5017450
                               H3k4me2    H3k4me3   H3k79me2     H3k9ac
ENSG00000000419.7.49575069   1.2136176  1.1202901  1.5155803  1.2468256
ENSG00000000457.8.169863093  0.6456572  0.6508561  0.7976487  0.5792891
ENSG00000000938.7.27961645  -0.1878255 -0.6560973 -1.3803974 -1.0067972
                              H3k9me1   H3k9me3   H4k20me1
ENSG00000000419.7.49575069  0.1426980  1.185622  1.9599992
ENSG00000000457.8.169863093 0.3630902  1.014923 -0.2695111
ENSG00000000938.7.27961645  0.6564520 -1.370871 -1.8773178
```

17.6.2. Create task

```
exp_pred_task = as_task_regr(exp_pred_data, target='gene_expression')
```

Partition the data into test and training sets. We will use $\frac{1}{3}$ and $\frac{2}{3}$ of the data for testing.

```
split = partition(exp_pred_task)
```

17.6.3. Example learners

17.6.3.1. Linear regression

```
learner = lrn("regr.lm")
```

17.6.3.1.1. Train

```
learner$train(exp_pred_task, split$train)
```

17.6.3.1.2. Predict

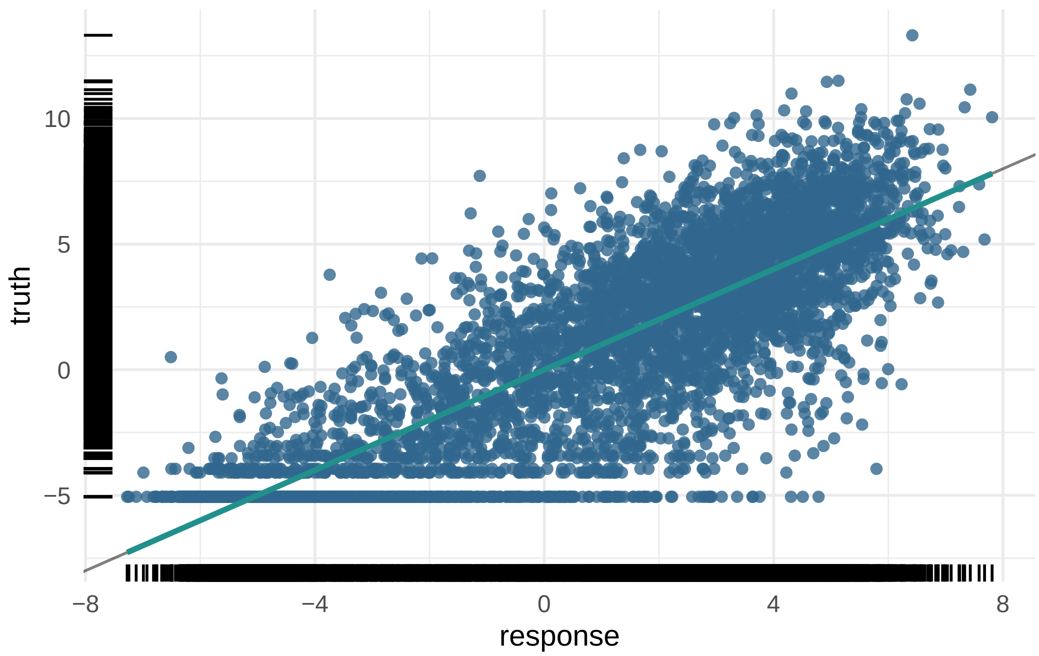
```
pred_train = learner$predict(exp_pred_task, split$train)
pred_test = learner$predict(exp_pred_task, split$test)
```

17.6.3.1.3. Assess

```
pred_train
```

```
<PredictionRegr> for 5789 observations:
    row_ids     truth response
          1  6.082343 5.139251
          2  2.989145 2.909552
          7  5.838076 4.563759
---
       8543  9.016443 6.141272
       8583  7.475697 2.543423
       8618 10.049236 5.523896
```

```
plot(pred_train)
```



For the training data:

```
measures = msrs(c('regr.rsq'))
pred_train$score(measures)
```

```
 regr.rsq
0.7495474
```

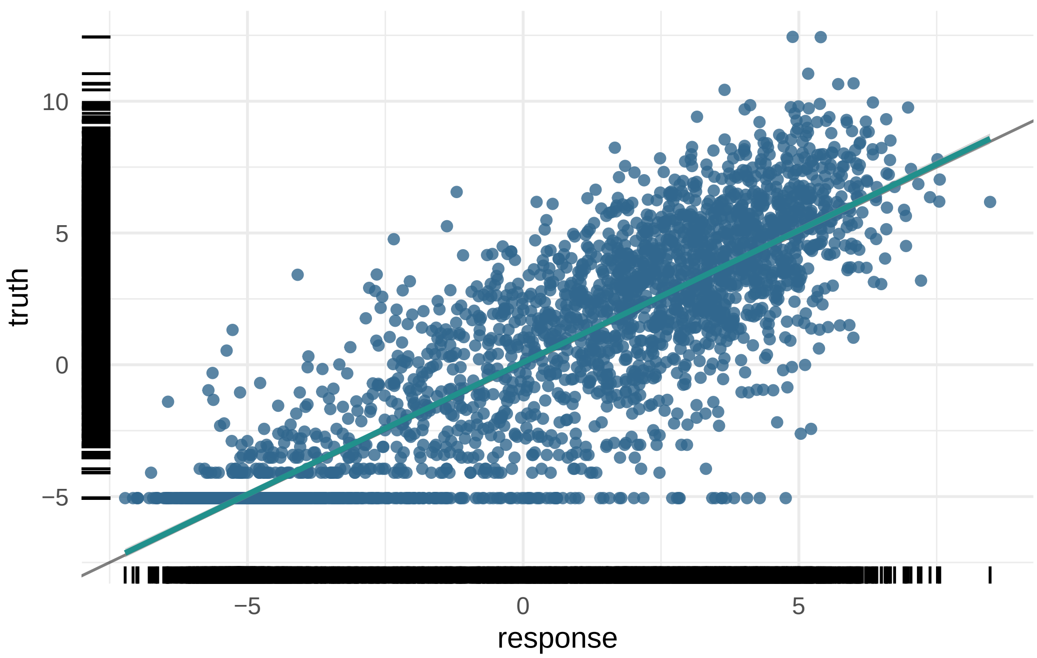
And the test data:

```
pred_test$score(measures)
```

```
 regr.rsq
0.7526609
```

And the plot of the test data predictions:

```
plot(pred_test)
```



17.6.3.2. Penalized regression

Imagine you want to teach a computer to predict house prices based on various features like size, number of bedrooms, and location. You decide to use **regression**, which finds a relationship between these features and the house prices. But what if your model becomes too complicated? This is where **penalized regression** comes in.

17.6.3.2.1. The Problem with Overfitting

Sometimes, the model tries too hard to fit every single data point perfectly. This can make the model very complex, like trying to draw a perfect line through a very bumpy path. This problem is called **overfitting**. An overfitted model works well on the data it has seen (training data) but performs poorly on new, unseen data (testing data).

17.6.3.2.2. Introducing Penalized Regression

Penalized regression helps prevent overfitting by adding a "penalty" to the model for being too complex. Think of it as a way to encourage the model to be simpler and more general. There are three common types of penalized regression:

1. **Ridge Regression (L2 Penalty):**
   - Adds a penalty based on the size of the coefficients. It tries to keep all coefficients small.
   - If the model's equation looks too complicated, Ridge Regression will push it towards a simpler form by shrinking the coefficients.
   - Imagine you have a rubber band that pulls the coefficients towards zero, making the model less likely to overfit.
2. **Lasso Regression (L1 Penalty):**
   - Adds a penalty that can shrink some coefficients all the way to zero.
   - This means Lasso Regression can completely remove some features from the model, making it simpler.
   - Imagine you have a pair of scissors that can cut off the least important features, leaving only the most important ones.
3. **Elastic Net:**
   - Combines both Ridge and Lasso penalties. It adds penalties for both the size and the number of coefficients.
   - This method balances between shrinking coefficients and eliminating some altogether, offering the benefits of both Ridge and Lasso.

- Think of Elastic Net as using both the rubber band (Ridge) and scissors (Lasso) to simplify the model.

With our data, the number of predictors is not huge, but we might be interested in 1) reducing overfitting, 2) improving interpretability, or 3) both by minimizing the number of predictors in our model without drastically affecting our prediction accuracy. Without penalized regression, the model might come up with a very complex equation. With Ridge, Lasso, or Elastic Net, the model simplifies this equation by either shrinking the coefficients (Ridge), removing some of them (Lasso), or balancing both (Elastic Net).

Here's a simple summary:

- **Ridge Regression:** Reduces the impact of less important features by shrinking their coefficients.
- **Lasso Regression:** Can eliminate some features entirely by setting their coefficients to zero.
- **Elastic Net:** Combines the effects of Ridge and Lasso, shrinking some coefficients and eliminating others.

Using penalized regression in machine learning ensures that your model:

1. **Performs Better on New Data:** By avoiding overfitting, the model can make more accurate predictions on new, unseen data.
2. **Is Easier to Interpret:** A simpler model with fewer features is easier to understand and explain.

17.6.3.3. Penalized Regression with mlr3

In the mlr3 package, you can easily apply penalized regression methods to your tasks. Here's how:

1. **Select Penalized Regression Learners:** mlr3 provides learners for Ridge, Lasso, and Elastic Net Regression.
2. **Train the Learner:** Use your data to train the chosen penalized regression model.
3. **Evaluate and Adjust:** Check how well the model performs and make adjustments if needed.

This description explains penalized regression, including Ridge, Lasso, and Elastic Net, in an intuitive way, highlighting their benefits and how they work, while relating them to familiar concepts and the mlr3 package.

Recall that we can use penalized regression to select the most important predictors from a large set of predictors. In this case, we will use the `glmnet` package to perform penalized regression, but we will use the `mlr` interface to `glmnet` to make it easier to use.

The `nfolds` parameter is the number of folds to use in the cross-validation procedure.

What is Cross-Validation? Cross-validation is a technique used to assess how well a model will perform on unseen data. It involves splitting the data into multiple parts, training the model on some of these parts, and validating it on the remaining parts. This process is repeated several times to ensure the model's performance is consistent.

Why Use Cross-Validation? Cross-validation helps to:

- Avoid Overfitting: By testing the model on different subsets of the data, cross-validation helps ensure that the model does not memorize the training data but learns to generalize from it.
- Select the Best Model Parameters: Penalized regression models, such as those trained with glmnet, have parameters that control the strength of the penalty (e.g., lambda). Cross-validation helps find the best values for these parameters.

When using the glmnet package, cross-validation can be performed using the cv.glmnet function. Here's how the process works:

1. Split the Data: The data is divided into k folds (common choices are 5 or 10 folds). Each fold is a subset of the data.

2. Train and Validate: The model is trained k times. In each iteration, -1 k-1 folds are used for training, and the remaining fold is used for validation. This process is repeated until each fold has been used as the validation set exactly once.

3. Calculate Performance: The performance of the model (e.g., mean squared error for regression) is calculated for each fold. The average performance across all folds is computed to get an overall measure of how well the model is expected to perform on unseen data.

4. Select the Best Parameters: The cv.glmnet function evaluates different values of the penalty parameter (lambda). It selects the lambda value that results in the best average performance across the folds.

In this case, we will use the `cv_glmnet` learner, which will automatically select the best value of the penalization parameters. When the `alpha` parameter is set to 0, the model is a Ridge regression model. When the `alpha` parameter is set to 1, the model is a Lasso regression model.

```r
learner = lrn("regr.cv_glmnet", nfolds=10, alpha=0)
```

17.6.3.3.1. Train

```r
learner$train(exp_pred_task)
```

```r
measures = msrs(c('regr.rsq', 'regr.mse', 'regr.rmse'))
pred_train$score(measures)
```

```
 regr.rsq  regr.mse regr.rmse 
0.7495474 4.8736194 2.2076275 
```

In the case of the penalized regression, we can also look at the coefficients of the model.

```r
coef(learner$model)
```

```
15 x 1 sparse Matrix of class "dgCMatrix"
                     s1
(Intercept)  0.10173828
Control     -0.08042502
Dnase        0.91127090
H2az         0.33880640
H3k27ac      0.15845313
H3k27me3    -0.25171391
H3k36me3     0.72063384
H3k4me1     -0.08222957
H3k4me2      0.13101892
H3k4me3      0.38905759
H3k79me2     0.99247076
H3k9ac       0.52009300
H3k9me1     -0.09183614
H3k9me3     -0.17912878
H4k20me1     0.11235659
```

Note that the coefficients are all zero for the histone marks that were not selected by the model. In this case, we can use the model not to predict new data, but to help us understand the data.

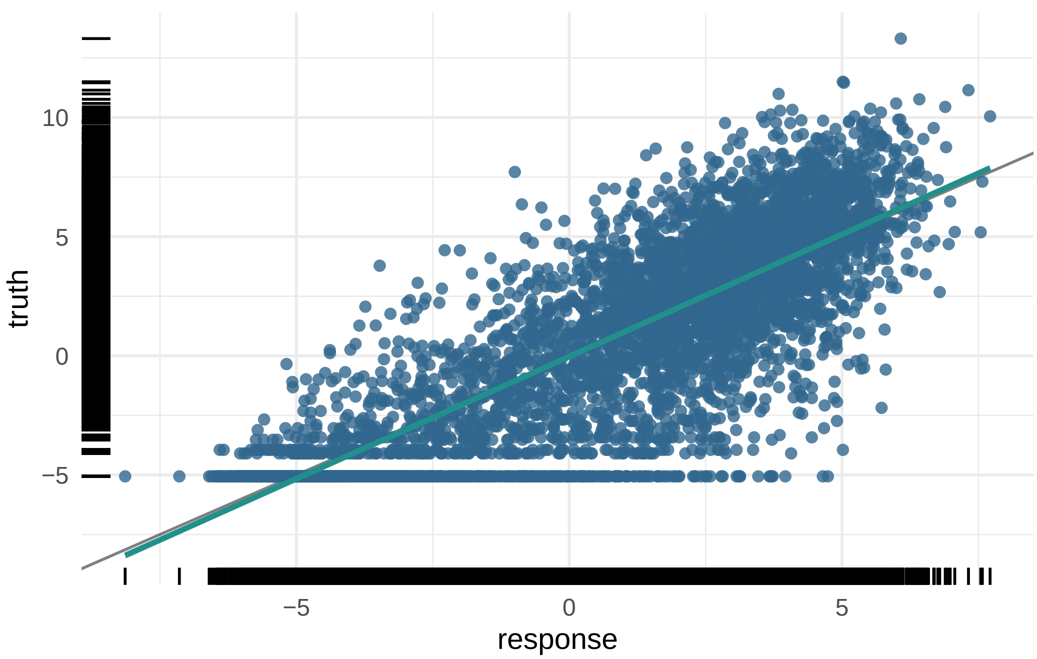
```
pred_train = learner$predict(exp_pred_task, split$train)
pred_test = learner$predict(exp_pred_task, split$test)
```

17.6.3.3.2. Assess

```
pred_train
```

```
<PredictionRegr> for 5789 observations:
    row_ids     truth response
          1  6.082343 4.923259
          2  2.989145 2.936421
          7  5.838076 4.619141
---
       8543  9.016443 5.580735
       8583  7.475697 2.565638
       8618 10.049236 5.226577
```

```
plot(pred_train)
```



For the training data:

```
measures = msrs(c('regr.rsq'))
pred_train$score(measures)
```

```
 regr.rsq
0.7422423
```

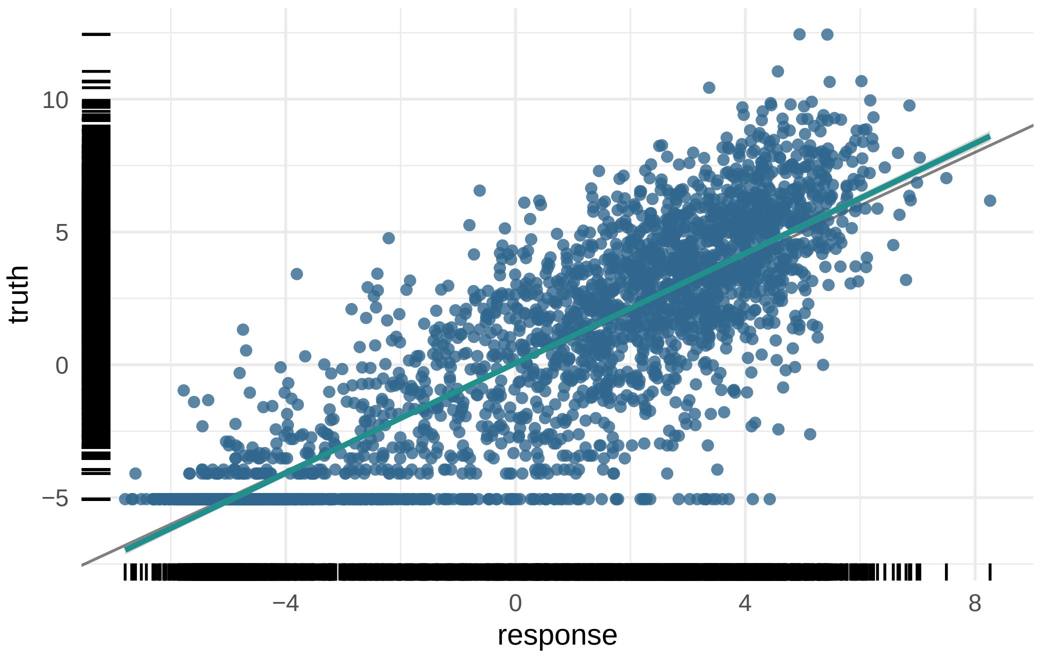
And the test data:

```
pred_test$score(measures)
```

```
 regr.rsq
0.7481403
```

And the plot of the test data predictions:

```
plot(pred_test)
```



```r
# Calculate the R-squared value
truth <- pred_test$truth
predicted <- pred_test$response
rss <- sum((truth - predicted)^2)  # Residual sum of squares
tss <- sum((truth - mean(truth))^2)  # Total sum of squares
r_squared <- 1 - (rss / tss)
```

Part V.

Bioconductor

18. Accessing and working with public omics data

18.1. Background

The data we are going to access are from this paper.

> Background: The tumor microenvironment is an important factor in cancer immunotherapy response. To further understand how a tumor affects the local immune system, we analyzed immune gene expression differences between matching normal and tumor tissue.Methods: We analyzed public and new gene expression data from solid cancers and isolated immune cell populations. We also determined the correlation between CD8, FoxP3 IHC, and our gene signatures.Results: We observed that regulatory T cells (Tregs) were one of the main drivers of immune gene expression differences between normal and tumor tissue. A tumor-specific CD8 signature was slightly lower in tumor tissue compared with normal of most (12 of 16) cancers, whereas a Treg signature was higher in tumor tissue of all cancers except liver. Clustering by Treg signature found two groups in colorectal cancer datasets. The high Treg cluster had more samples that were consensus molecular subtype 1/4, right-sided, and microsatellite-instable, compared with the low Treg cluster. Finally, we found that the correlation between signature and IHC was low in our small dataset, but samples in the high Treg cluster had significantly more CD8+ and FoxP3+ cells compared with the low Treg cluster.Conclusions: Treg gene expression is highly indicative of the overall tumor immune environment.Impact: In comparison with the consensus molecular subtype and microsatellite status, the Treg signature identifies more colorectal tumors with high immune activation that may benefit from cancer immunotherapy.

In this little exercise, we will:

1. Access public omics data using the GEOquery package
2. Get an opportunity to work with another SummarizedExperiment object.
3. Perform a simple unsupervised analysis to visualize these public data.

18.2. GEOquery to PCA

The first step is to install the R package GEOquery. This package allows us to access data from the Gene Expression Omnibus (GEO) database. GEO is a public repository of omics data.

```
BiocManager::install("GEOquery")
```

GEOquery has only one commonly used function, `getGEO()` which takes a GEO accession number as an argument. The GEO accession number is a unique identifier for a dataset.

Use the GEOquery package to fetch data about GSE103512.

```
library(GEOquery)
gse = getGEO("GSE103512")[[1]]
```

You might ask why we are using `[[1]]` at the end of the `getGEO()` function. The reason is that `getGEO()` returns a list of `GSE` objects. We are only interested in the first one (and in this case, the only one). We return a list of `GSE` objects because in the early days, it was not unusual to have a single GEO accession number represent multiple datasets. While uncommon now, we've kept the convention since lots of "older" data is still quite useful.

Again, a historically-derived detail, is to convert from the older Bioconductor data structure (GEOquery was written in 2007), the `ExpressionSet`, to the newer `SummarizedExperiment`.

```
library(SummarizedExperiment)
se = as(gse, "SummarizedExperiment")
```

Use some code to determine the answers to the following:

- What is the class of `se`?
- What are the dimensions of `se`?
- What are the dimensions of the `assay` slot of `se`?
- What are the dimensions of the `colData` slot of `se`?
- What variables are in the `colData` slot of `se`?

Examine two variables of interest, cancer type and tumor/normal status. The `with` function is a convenience to allow us to access variables in a data frame by name (rather than having to do `dataframe$variable_name`. Recalling that the `table` function is a convenient way to summarize the counts of unique values in a vector, we can use `with` to access the variables of interest and `table` to summarize the counts of unique values.

```
with(colData(se),table(`cancer.type.ch1`,`normal.ch1`))
```

```
               normal.ch1
cancer.type.ch1 no yes
          BC    65  10
          CRC   57  12
          NSCLC 60   9
          PCA   60   7
```

- How many samples are there of each cancer type?
- How many samples are there of each tumor/normal status?

When performing unsupervised analysis, it is common to filter genes by variance to find the most informative genes. It is common practice to filter genes by standard deviation or some other measure of variability and keep the top X percent of them when performing dimensionality reduction. There is not a single right answer to what percentage to use, so try a few to see what happens. In the example code, I chose to use the top 500 genes by standard deviation, but you can play with the threshold to see what happens.

Recall that the `assay` function is used to access the data matrix of the `SummarizedExperiment` object.

Think through the code below and then run it.

```
sds = apply(assay(se, 'exprs'),1,sd)
dat = assay(se, 'exprs')[order(sds,decreasing = TRUE)[1:500],]
```

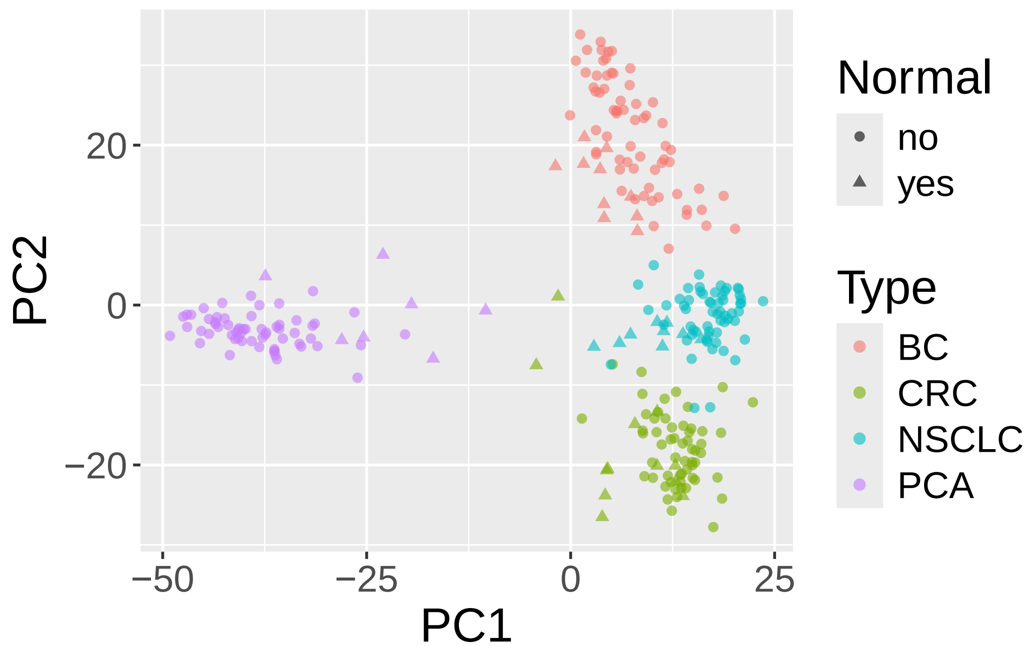
If you don't recognize the function `apply`, it is a function that applies a function to each row or column of a matrix. In this case, we are applying the `sd` function to each row of the data matrix. The `order` function is used to sort the standard deviations in decreasing order (when `decreasing=TRUE`). And the `[1:500]` is used to subset the data matrix to the top 500 genes by standard deviation.

Perform PCA and prepare for plotting. We will be using ggplot2, so we need to make a data.frame before plotting.

```
pca_results <- prcomp(t(dat))
pca_df = as.data.frame(pca_results$x)
pca_df$Type=factor(colData(se)[,'cancer.type.ch1'])
pca_df$Normal = factor(colData(se)[,'normal.ch1'])
```

Now, we are going to plot the results of the PCA, coloring the points by cancer type and using different shapes for normal and tumor samples.

```
library(ggplot2)
ggplot(pca_df, aes(x=PC1,y=PC2,shape=Normal,color=Type)) +
    geom_point( alpha=0.6) + theme(text=element_text(size = 18))
```



In this case, the x-axis is the first principal component and the y-axis is the second principal component.

- What do you see?
- What about additional principal components?
- Bonus: Try using the `GGally` package to plot principal components (using the `ggpairs` function).
- Bonus: Calculate the variance explained by each principal component and plot the results.

19. Introduction to SummarizedExperiment

The `SummarizedExperiment` class is used to store rectangular matrices of experimental results, which are commonly produced by sequencing and microarray experiments. Each object stores observations of one or more samples, along with additional meta-data describing both the observations (features) and samples (phenotypes).

A key aspect of the `SummarizedExperiment` class is the coordination of the meta-data and assays when subsetting. For example, if you want to exclude a given sample you can do for both the meta-data and assay in one operation, which ensures the meta-data and observed data will remain in sync. Improperly accounting for meta and observational data has resulted in a number of incorrect results and retractions so this is a very desirable property.

`SummarizedExperiment` is in many ways similar to the historical `ExpressionSet`, the main distinction being that `SummarizedExperiment` is more flexible in it's row information, allowing both `GRanges` based as well as those described by arbitrary `DataFrame`s. This makes it ideally suited to a variety of experiments, particularly sequencing based experiments such as RNA-Seq and ChIp-Seq.

```
BiocManager::install('airway')
BiocManager::install('SummarizedExperiment')
```

19.1. Anatomy of a SummarizedExperiment

The *SummarizedExperiment* package contains two classes: `SummarizedExperiment` and `RangedSummarizedExperiment`.

`SummarizedExperiment` is a matrix-like container where rows represent features of interest (e.g. genes, transcripts, exons, etc.) and columns represent samples. The objects contain one or more assays, each represented by a matrix-like object of numeric or other mode. The rows of a `SummarizedExperiment` object represent features of interest. Information about these features is stored in a `DataFrame` object, accessible using the function `rowData()`. Each row of the `DataFrame` provides information on the feature in the corresponding row of the `SummarizedExperiment` object. Columns of the DataFrame represent different attributes of the features of interest, e.g., gene or transcript IDs, etc.

`RangedSummarizedExperiment` is the “child”” of the `SummarizedExperiment` class which means that all the methods on `SummarizedExperiment` also work on a `RangedSummarizedExperiment`.

The fundamental difference between the two classes is that the rows of a `RangedSummarizedExperiment` object represent genomic ranges of interest instead of a `DataFrame` of features. The `RangedSummarizedExperiment` ranges are described by a `GRanges` or a `GRangesList` object, accessible using the `rowRanges()` function.

Figure 19.1 displays the class geometry and highlights the vertical (column) and horizontal (row) relationships.

19.1.1. Assays

The `airway` package contains an example dataset from an RNA-Seq experiment of read counts per gene for airway smooth muscles. These data are stored in a `RangedSummarizedExperiment` object which contains 8 different experimental and assays 64,102 gene transcripts.

```
Loading required package: airway
```

```
library(SummarizedExperiment)
data(airway, package="airway")
se <- airway
se
```

```
class: RangedSummarizedExperiment
dim: 63677 8
metadata(1): ''
assays(1): counts
rownames(63677): ENSG00000000003 ENSG00000000005 ... ENSG00000273492
  ENSG00000273493
rowData names(10): gene_id gene_name ... seq_coord_system symbol
colnames(8): SRR1039508 SRR1039509 ... SRR1039520 SRR1039521
colData names(9): SampleName cell ... Sample BioSample
```

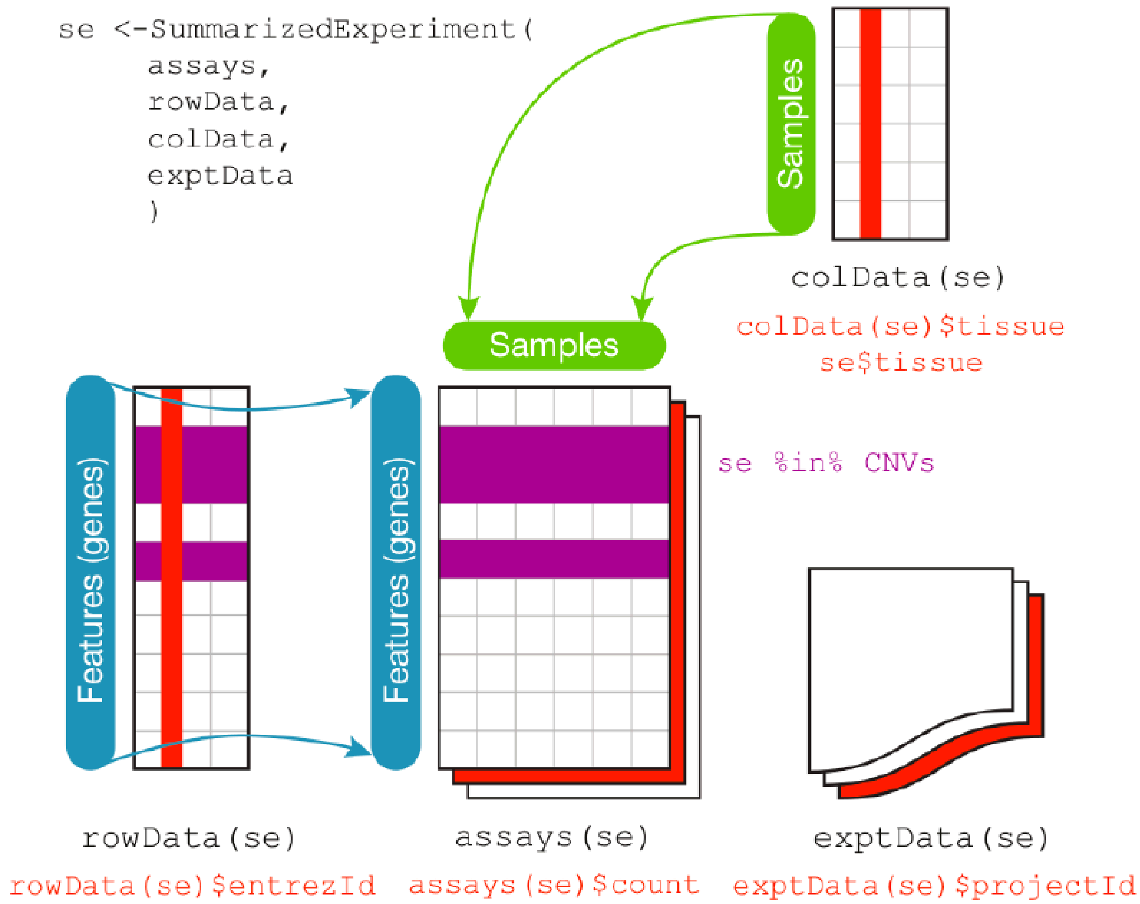


Figure 19.1.: Summarized Experiment. There are three main components, the `colData()`, the `rowData()` and the `assays()`. The accessors for the various parts of a complete SummarizedExperiment object match the names.

To retrieve the experiment data from a `SummarizedExperiment` object one can use the `assays()` accessor. An object can have multiple assay datasets each of which can be accessed using the `$` operator. The `airway` dataset contains only one assay (`counts`). Here each row represents a gene transcript and each column one of the samples.

```
assays(se)$counts
```

	SRR1039508	SRR1039509	SRR1039512	SRR1039513	SRR1039516	SRR1039517	SRR1039520	SRR1039521
ENSG00000000003	679	448	873	408	1138	1047	770	572
ENSG00000000005	0	0	0	0	0	0	0	0
ENSG00000000419	467	515	621	365	587	799	417	508
ENSG00000000457	260	211	263	164	245	331	233	229
ENSG00000000460	60	55	40	35	78	63	76	60
ENSG00000000938	0	0	2	0	1	0	0	0
ENSG00000000971	3251	3679	6177	4252	6721	11027	5176	7995
ENSG00000001036	1433	1062	1733	881	1424	1439	1359	1109
ENSG00000001084	519	380	595	493	820	714	696	704
ENSG00000001167	394	236	464	175	658	584	360	269

19.1.2. 'Row' (regions-of-interest) data

The `rowRanges()` accessor is used to view the range information for a `RangedSummarizedExperiment`. (Note if this were the parent `SummarizedExperiment` class we'd use `rowData()`). The data are stored in a `GRangesList` object, where each list element corresponds to one gene transcript and the ranges in each `GRanges` correspond to the exons in the transcript.

```
rowRanges(se)
```

```
GRangesList object of length 63677:
$ENSG00000000003
GRanges object with 17 ranges and 2 metadata columns:
       seqnames            ranges strand |   exon_id       exon_name
          <Rle>         <IRanges>  <Rle> | <integer>     <character>
   [1]        X 99883667-99884983      - |    667145 ENSE00001459322
   [2]        X 99885756-99885863      - |    667146 ENSE00000868868
   [3]        X 99887482-99887565      - |    667147 ENSE00000401072
   [4]        X 99887538-99887565      - |    667148 ENSE00001849132
   [5]        X 99888402-99888536      - |    667149 ENSE00003554016
```

```
  ...        ...                ...     ... .        ...               ...
  [13]         X 99890555-99890743       - |    667156 ENSE00003512331
  [14]         X 99891188-99891686       - |    667158 ENSE00001886883
  [15]         X 99891605-99891803       - |    667159 ENSE00001855382
  [16]         X 99891790-99892101       - |    667160 ENSE00001863395
  [17]         X 99894942-99894988       - |    667161 ENSE00001828996
  -------
  seqinfo: 722 sequences (1 circular) from an unspecified genome

...
<63676 more elements>
```

19.1.3. 'Column' (sample) data

Sample meta-data describing the samples can be accessed using `colData()`, and is a `DataFrame` that can store any number of descriptive columns for each sample row.

```r
colData(se)
```

```
DataFrame with 8 rows and 9 columns
           SampleName     cell      dex    albut        Run avgLength
             <factor> <factor> <factor> <factor>   <factor> <integer>
SRR1039508 GSM1275862  N61311     untrt    untrt SRR1039508       126
SRR1039509 GSM1275863  N61311     trt      untrt SRR1039509       126
SRR1039512 GSM1275866  N052611    untrt    untrt SRR1039512       126
SRR1039513 GSM1275867  N052611    trt      untrt SRR1039513        87
SRR1039516 GSM1275870  N080611    untrt    untrt SRR1039516       120
SRR1039517 GSM1275871  N080611    trt      untrt SRR1039517       126
SRR1039520 GSM1275874  N061011    untrt    untrt SRR1039520       101
SRR1039521 GSM1275875  N061011    trt      untrt SRR1039521        98
           Experiment    Sample    BioSample
             <factor>  <factor>     <factor>
SRR1039508  SRX384345 SRS508568 SAMN02422669
SRR1039509  SRX384346 SRS508567 SAMN02422675
SRR1039512  SRX384349 SRS508571 SAMN02422678
SRR1039513  SRX384350 SRS508572 SAMN02422670
SRR1039516  SRX384353 SRS508575 SAMN02422682
SRR1039517  SRX384354 SRS508576 SAMN02422673
SRR1039520  SRX384357 SRS508579 SAMN02422683
SRR1039521  SRX384358 SRS508580 SAMN02422677
```

This sample metadata can be accessed using the $ accessor which makes it easy to subset the entire object by a given phenotype.

```
# subset for only those samples treated with dexamethasone
se[, se$dex == "trt"]
```

```
class: RangedSummarizedExperiment
dim: 63677 4
metadata(1): ''
assays(1): counts
rownames(63677): ENSG00000000003 ENSG00000000005 ... ENSG00000273492
  ENSG00000273493
rowData names(10): gene_id gene_name ... seq_coord_system symbol
colnames(4): SRR1039509 SRR1039513 SRR1039517 SRR1039521
colData names(9): SampleName cell ... Sample BioSample
```

19.1.4. Experiment-wide metadata

Meta-data describing the experimental methods and publication references can be accessed using `metadata()`.

```
metadata(se)
```

```
[[1]]
Experiment data
  Experimenter name: Himes BE
  Laboratory: NA
  Contact information:
  Title: RNA-Seq transcriptome profiling identifies CRISPLD2 as a glucocorticoid responsive
  URL: http://www.ncbi.nlm.nih.gov/pubmed/24926665
  PMIDs: 24926665

  Abstract: A 226 word abstract is available. Use 'abstract' method.
```

Note that `metadata()` is just a simple list, so it is appropriate for *any* experiment wide metadata the user wishes to save, such as storing model formulas.

```r
metadata(se)$formula <- counts ~ dex + albut

metadata(se)
```

```
[[1]]
Experiment data
  Experimenter name: Himes BE
  Laboratory: NA
  Contact information:
  Title: RNA-Seq transcriptome profiling identifies CRISPLD2 as a glucocorticoid responsive
  URL: http://www.ncbi.nlm.nih.gov/pubmed/24926665
  PMIDs: 24926665

  Abstract: A 226 word abstract is available. Use 'abstract' method.

$formula
counts ~ dex + albut
```

19.2. Common operations on `SummarizedExperiment`

19.2.1. Subsetting

- `[` Performs two dimensional subsetting, just like subsetting a matrix or data frame.

```r
# subset the first five transcripts and first three samples
se[1:5, 1:3]
```

```
class: RangedSummarizedExperiment
dim: 5 3
metadata(2): '' formula
assays(1): counts
rownames(5): ENSG00000000003 ENSG00000000005 ENSG00000000419
  ENSG00000000457 ENSG00000000460
rowData names(10): gene_id gene_name ... seq_coord_system symbol
colnames(3): SRR1039508 SRR1039509 SRR1039512
colData names(9): SampleName cell ... Sample BioSample
```

- `$` operates on `colData()` columns, for easy sample extraction.

```
se[, se$cell == "N61311"]
```

```
class: RangedSummarizedExperiment
dim: 63677 2
metadata(2): '' formula
assays(1): counts
rownames(63677): ENSG00000000003 ENSG00000000005 ... ENSG00000273492
  ENSG00000273493
rowData names(10): gene_id gene_name ... seq_coord_system symbol
colnames(2): SRR1039508 SRR1039509
colData names(9): SampleName cell ... Sample BioSample
```

19.2.2. Getters and setters

- `rowRanges()` / (`rowData()`), `colData()`, `metadata()`

```
counts <- matrix(1:15, 5, 3, dimnames=list(LETTERS[1:5], LETTERS[1:3]))

dates <- SummarizedExperiment(assays=list(counts=counts),
                              rowData=DataFrame(month=month.name[1:5], day=1:5))

# Subset all January assays
dates[rowData(dates)$month == "January", ]
```

```
class: SummarizedExperiment
dim: 1 3
metadata(0):
assays(1): counts
rownames(1): A
rowData names(2): month day
colnames(3): A B C
colData names(0):
```

- `assay()` versus `assays()` There are two accessor functions for extracting the assay data from a `SummarizedExperiment` object. `assays()` operates on the entire list of assay data as a whole, while `assay()` operates on only one assay at a time. `assay(x, i)` is simply a convenience function which is equivalent to `assays(x)[[i]]`.

```r
assays(se)
```

```
List of length 1
names(1): counts
```

```r
assays(se)[[1]][1:5, 1:5]
```

```
                SRR1039508 SRR1039509 SRR1039512 SRR1039513 SRR1039516
ENSG00000000003        679        448        873        408       1138
ENSG00000000005          0          0          0          0          0
ENSG00000000419        467        515        621        365        587
ENSG00000000457        260        211        263        164        245
ENSG00000000460         60         55         40         35         78
```

```r
# assay defaults to the first assay if no i is given
assay(se)[1:5, 1:5]
```

```
                SRR1039508 SRR1039509 SRR1039512 SRR1039513 SRR1039516
ENSG00000000003        679        448        873        408       1138
ENSG00000000005          0          0          0          0          0
ENSG00000000419        467        515        621        365        587
ENSG00000000457        260        211        263        164        245
ENSG00000000460         60         55         40         35         78
```

```r
assay(se, 1)[1:5, 1:5]
```

```
                SRR1039508 SRR1039509 SRR1039512 SRR1039513 SRR1039516
ENSG00000000003        679        448        873        408       1138
ENSG00000000005          0          0          0          0          0
ENSG00000000419        467        515        621        365        587
ENSG00000000457        260        211        263        164        245
ENSG00000000460         60         55         40         35         78
```

19.2.3. Range-based operations

- `subsetByOverlaps()` `SummarizedExperiment` objects support all of the `findOverlaps()` methods and associated functions. This includes `subsetByOverlaps()`, which makes it easy to subset a `SummarizedExperiment` object by an interval.

In tne next code block, we define a region of interest (or many regions of interest) and then subset our `SummarizedExperiment` by overlaps with this region.

```
# Subset for only rows which are in the interval 100,000 to 110,000 of
# chromosome 1
roi <- GRanges(seqnames="1", ranges=100000:1100000)
sub_se = subsetByOverlaps(se, roi)
sub_se
```

```
class: RangedSummarizedExperiment
dim: 74 8
metadata(2): '' formula
assays(1): counts
rownames(74): ENSG00000131591 ENSG00000177757 ... ENSG00000272512
  ENSG00000273443
rowData names(10): gene_id gene_name ... seq_coord_system symbol
colnames(8): SRR1039508 SRR1039509 ... SRR1039520 SRR1039521
colData names(9): SampleName cell ... Sample BioSample
```

```
dim(sub_se)
```

```
[1] 74  8
```

19.3. Constructing a SummarizedExperiment

Often, `SummarizedExperiment` or `RangedSummarizedExperiment` objects are returned by functions written by other packages. However it is possible to create them by hand with a call to the `SummarizedExperiment()` constructor. The code below is simply to illustrate the mechanics of creating an object from scratch. In practice, you will probably have the pieces of the object from other sources such as Excel files or csv files.

Constructing a `RangedSummarizedExperiment` with a `GRanges` as the *rowRanges* argument:

```
nrows <- 200
ncols <- 6
counts <- matrix(runif(nrows * ncols, 1, 1e4), nrows)
rowRanges <- GRanges(rep(c("chr1", "chr2"), c(50, 150)),
                     IRanges(floor(runif(200, 1e5, 1e6)), width=100),
                     strand=sample(c("+", "-"), 200, TRUE),
                     feature_id=sprintf("ID%03d", 1:200))
colData <- DataFrame(Treatment=rep(c("ChIP", "Input"), 3),
                     row.names=LETTERS[1:6])

SummarizedExperiment(assays=list(counts=counts),
                     rowRanges=rowRanges, colData=colData)
```

```
class: RangedSummarizedExperiment
dim: 200 6
metadata(0):
assays(1): counts
rownames: NULL
rowData names(1): feature_id
colnames(6): A B ... E F
colData names(1): Treatment
```

A `SummarizedExperiment` can be constructed with or without supplying a `DataFrame` for the *rowData* argument:

```
SummarizedExperiment(assays=list(counts=counts), colData=colData)
```

```
class: SummarizedExperiment
dim: 200 6
metadata(0):
assays(1): counts
rownames: NULL
rowData names(0):
colnames(6): A B ... E F
colData names(1): Treatment
```

20. Ranges Exercises

In the following exercises, we will use the `GenomicRanges` package to explore range operations. We will use the `AnnotationHub` package to load DNAse hypersensitivity data from the ENCODE project. In practice, the ENCODE project published datasets like these as bed files. AnnotationHub has packaged these into GRanges objects that we can load and use directly. However, if you have a bed file of your own (peak calls, enhancer regions, etc.), you can load them into GRanges objects using `rtracklayer::import`.

20.1. Exercise 1

In this exercise, we will use DNAse hypersensitivity data to practice working with a GRanges object.

- Use the `AnnotationHub` package to find the `goldenpath/hg19/encodeDCC/wgEncodeUwDnase/wgEncodeU` `GRanges` object. Load that into R as the variable dnase.

```
library(AnnotationHub)
ah = AnnotationHub()
query(ah, "goldenpath/hg19/encodeDCC/wgEncodeUwDnase/wgEncodeUwDnaseK562PkRep1.narrowPeak.g
# the thing above should have only one record, so we can
# just grab it
dnase = query(ah, "goldenpath/hg19/encodeDCC/wgEncodeUwDnase/wgEncodeUwDnaseK562PkRep1.narr
```

- What type of object is dnase?

```
dnase
class(dnase)
```

- What metadata is stored in dnase?

```
mcols(dnase)
```

- How many peaks are on each chromosome?

```
library(GenomicFeatures)
table(seqnames(dnase))
```

- What are the mean, min, max, and median widths of the peaks?

```
summary(width(dnase))
```

- What are the sequences that were used in the analysis? Do the names have "chr" or not? Experiment with changing the `seqlevelsStyle` to adjust the sequence names.

```
seqlevels(dnase)
seqlevelsStyle(dnase)
seqlevelsStyle(dnase) = 'ensembl'
seqlevelsStyle(dnase)
seqlevels(dnase)
```

- What is the total amount of "landscape" covered by the peaks? Assume that the peaks do not overlap. What portion of the genome does this represent?

```
sum(width(dnase))
sum(seqlengths(dnase))
sum(width(dnase))/sum(seqlengths(dnase))
```

20.2. Exercise 2

In this exercise, we are going to load the second DNAse hypersensitivity replicate to investigate overlaps with the first replicate.

- Use the AnnotationHub to find the second replicate, `goldenpath/hg19/encodeDCC/wgEncodeUwDnase/wg` Load that as `dnase2`.

```
query(ah, "goldenpath/hg19/encodeDCC/wgEncodeUwDnase/wgEncodeUwDnaseK562PkRep2.narrowPeak.g
# the thing above should have only one record, so we can
# just grab it
dnase2 = query(ah, "goldenpath/hg19/encodeDCC/wgEncodeUwDnase/wgEncodeUwDnaseK562PkRep2.nar
```

- How many peaks are there in `dnase` and `dnase2`? Are there are similar number?

```
length(dnase)
length(dnase2)
```

- What are the peak sizes for `dnase2`?

```
summary(width(dnase2))
```

- What proportion of the genome does `dnase2` cover?

```
sum(width(dnase))/sum(seqlengths(dnase))
```

- Count the number of peaks from `dnase` that overlap with `dnase2`.

```
sum(dnase %over% dnase2)
```

- Assume that your peak-caller was "too specific" and that you want to expand your peaks by 50 bp on each end (so make them 100 bp larger). Use a combination of `resize` (and pay attention to the `fix` argument) and `width` to do this expansion to dnase and call the new `GRanges` object "dnase_wide".

```
w = width(dnase)
dnase_wide = resize(dnase, width=w+100, fix='center') #make a copy
width(dnase_wide)
```

20.3. Exercise 3

In this exercise, we are going to look at the overlap of DNAse sites relative to genes. To get started, install and load the `TxDb.Hsapiens.UCSC.hg19.knownGene` txdb object.

```
BiocManager::install("TxDb.Hsapiens.UCSC.hg19.knownGene")
library("TxDb.Hsapiens.UCSC.hg19.knownGene")
kg = TxDb.Hsapiens.UCSC.hg19.knownGene
```

- Load the transcripts from the knownGene txdb into a variable. What is the class of this object?

```
library("TxDb.Hsapiens.UCSC.hg19.knownGene")
kg = TxDb.Hsapiens.UCSC.hg19.knownGene
gx = genes(kg)
class(gx)
length(gx)
```

- Read about the `flank` method for GRanges objects. How could you use that to get the "promoter" regions of the transcripts? Let's assume that the promoter region is 2kb upstream of the gene.

```
flank(gx,2000)
```

- Instead of using flank, could you do the same thing with the TxDb object? (See `?promoters`).

```
proms = promoters(kg)
```

- Do any of the regions in the promoters overlap with each other?

```
summary(countOverlaps(proms))
```

- To find overlap of our DNAse sites with promoters, let's collapse overlapping "promoters" to just keep the contiguous regions by using `reduce`.

```
# reduce takes all overlapping regions and collapses them
# into a single region that spans all of the overlapping regions
prom_regions = reduce(proms)

# now we can check for overlaps
summary(countOverlaps(prom_regions))
```

- Count the number of DNAse sites that overlap with our promoter regions.

```
sum(dnase %over% prom_regions)
# if you notice no overlap, check the seqlevels
# and seqlevelsStyle
seqlevelsStyle(dnase) = "UCSC"
sum(dnase %over% prom_regions)
sum(dnase2 %over% prom_regions)
```

- Is this surprising? If we were to assume that the promoter and dnase regions are "independent" of each other, what number of overlaps would we expect?

```
prop_proms = sum(width(prom_regions))/sum(seqlengths(prom_regions))
prop_dnase = sum(width(dnase))/sum(seqlengths(prom_regions))
# Iff the dnase and promoter regions are
# not related, then we would expect this number
# of DNAse overlaps with promoters.
prop_proms * prop_dnase * length(dnase)
```

20.4. Exercise 4

We'll be using data from histone modification ChIP-seq experiments in human cells to illustrate the concepts of genomic ranges and features. The data consists of genomic intervals representing regions of the genome where specific histone modifications are enriched. These intervals are typically identified using ChIP-seq, a technique that maps protein-DNA interactions across the genome.

The ChIP-seq data is stored in a BED file format, which is a tab-delimited text file format commonly used to represent genomic intervals. Each line in the BED file corresponds to a genomic interval and contains information about the chromosome, start and end positions, and strand orientation of the interval. Additional columns may include metadata such as the signal strength or significance of the interval.

The AnnotationHub package in Bioconductor provides access to a wide range of genomic datasets, including ChIP-seq data. We can use this package to retrieve the ChIP-seq data for histone modifications in human cells and convert it into a GenomicRanges object for further analysis.

https://www.encodeproject.org/chip-seq/histone/

Let's start by loading the AnnotationHub package and retrieving the ChIP-seq data for histone modifications in human cells. You can read more about the AnnotationHub package and how to use it in the Bioconductor documentation.

```
library(AnnotationHub)
ah <- AnnotationHub()
```

There are multiple ways to search the AnnotationHub database. We've done that for you and here are the `GRanges` objects for each of four histone marks, and one histone mark replicate.

```
h3k4me1 <- ah[['AH25832']]
h3k4me3 <- ah[['AH25833']]
h3k9ac <- ah[['AH25834']]
h3k27me3 <- ah[['AH25835']]
h3k4me3_2 <- ah[['AH27284']]
```

Each of these variables now represents the peak calls after a chip-seq experiment pulling down the histone mark of interest. In the encode project these records were `bed` files. The bed files have been converted to `GRanges` objects to allow computation within R.

```
# Grab cpg islands as well
cpg = query(ah, c('cpg','UCSC','hg19'))[[1]]
```

Let's say that we don't know the behavior of the histone methylation marks with respect to CpG islands. We could ask the question, "What is the overlap of the histone peaks with CpG islands?"

```
sum(h3k4me1 %over% cpg)
```

We might want to actually count the number of bases of overlap between the methyl mark and CpG islands.

```
# The intersection of two peak sets results in the
# overlapping regions as a new set of regions
# The width of each peak is the number of overlapping bases
# And the sum of the widths is the total bases overlapping
sum(width(intersect(h3k4me1, cpg)))
```

But some methyl marks are known to have very broad signals, meaning that there is a higher chance of overlapping CpG islands just because there are more methylated bases. We can adjust for this by "normalizing" for all possible bases covered by either set of peaks, using `union`. We might think of this as a sort of "enrichment score" of one set in another set.

```
sum(width(union(h3k4me1, cpg)))
# and now "normalize"
sum(width(intersect(h3k4me1, cpg)))/sum(width(union(h3k4me1, cpg)))
```

Let's write a small function to calculate our little enrichment score.

```
range_enrichment_score <- function(r1, r2) {
  i = sum(width(intersect(r1, r2)))
  u = sum(width(union(r1,r2)))
  return(i/u)
}
```

And give it a try:

```
range_enrichment_score(h3k4me1, cpg)
```

21. ATAC-Seq with Bioconductor

Overview

Pre-requisites

This workshop assumes:

- A working and up-to-date version of R
- Basic knowledge of R syntax
- Familiarity with the *GenomicRanges* package and range manipulations
- Familiarity with BAM files and their contents

Participation

After a very brief review of ATAC-Seq and chromatin accessibility, students will work independently to follow this workflow. Additional materials are provided as links at the end of the workshop for those wanting deeper exposure. Additional materials include alignment from FASTQ files and peak calling.

R / *Bioconductor* packages used

- *Rsamtools*
- *GenomicRanges*
- *GenomicFeatures*
- *GenomicAlignments*
- *rtracklayer*
- *heatmaps*

Time outline

An example for a 45-minute workshop:

Activity	Time
Introduction	15m
Independent work	2-3hr
Additional exercises (optional, external)	up to 12 hours

Learning goals

- Describe how to import sequence alignments in BAM format into R
- Relate fragment size to genomic characteristics such as nucleosome occupancy and open chromatin.
- Perform basic alignment manipulations in R to enrich ATAC-seq data for chromatin characteristics.
- Gain familiarity with the IGV genome browser and examining data in genomic context.
- Visualize summaries of genomic signal using profile plots and heatmaps.

Learning objectives

- Load and save genomic data in BAM and BigWig formats [GenomicAlignments and rtracklayer].
- Perform basic QC plots from ATAC-Seq data.
- Isolate nucleosome-free and mononucleosome regions from ATAC-seq data.
- Install and use IGV to visualize data in genomic context.
- Create profile plots using the heatmaps package.

22. Background

Chromatin accessibility assays measure the extent to which DNA is open and accessible. Such assays now use high throughput sequencing as a quantitative readout. DNAse assays, first using microarrays(Crawford, Davis, et al. 2006) and then DNAse-Seq (Crawford, Holt, et al. 2006), requires a larger amount of DNA and is labor-indensive and has been largely supplanted by ATAC-Seq (Buenrostro et al. 2013).

The Assay for Transposase Accessible Chromatin with high-throughput sequencing (ATAC-seq) method maps chromatin accessibility genome-wide. This method quantifies DNA accessibility with a hyperactive Tn5 transposase that cuts and inserts sequencing adapters into regions of chromatin that are accessible. High throughput sequencing of fragments produced by the process map to regions of increased accessibility, transcription factor binding sites, and nucleosome positioning. The method is both fast and sensitive and can be used as a replacement for DNAse *and* MNase.

An early review of chromatin accessibility assays (Tsompana and Buck 2014) compares the use cases, pros and cons, and expected signals from each of the most common approaches (Figure @ref(fig:chromatinAssays)).

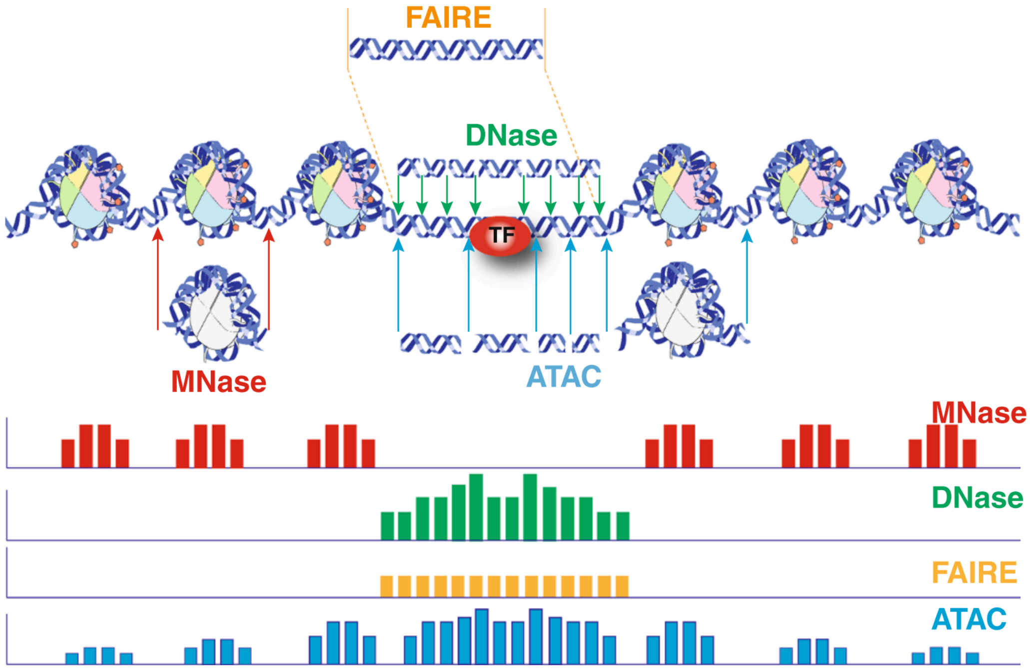


Figure 22.1.: Chromatin accessibility methods, compared. Representative DNA fragments generated by each assay are shown, with end locations within chromatin defined by colored arrows. Bar diagrams represent data signal obtained from each assay across the entire region. The footprint created by a transcription factor (TF) is shown for ATAC-seq and DNase-seq experiments.

The first manuscript describing ATAC-Seq protocol and findings outlined how ATAC-Seq data "line up" with other datatypes such as ChIP-seq and DNAse-seq (Figure @ref(fig:greenleaf)). They also highlight how fragment length correlates with specific genomic regions and characteristics (Buenrostro et al. 2013, fig. 3).

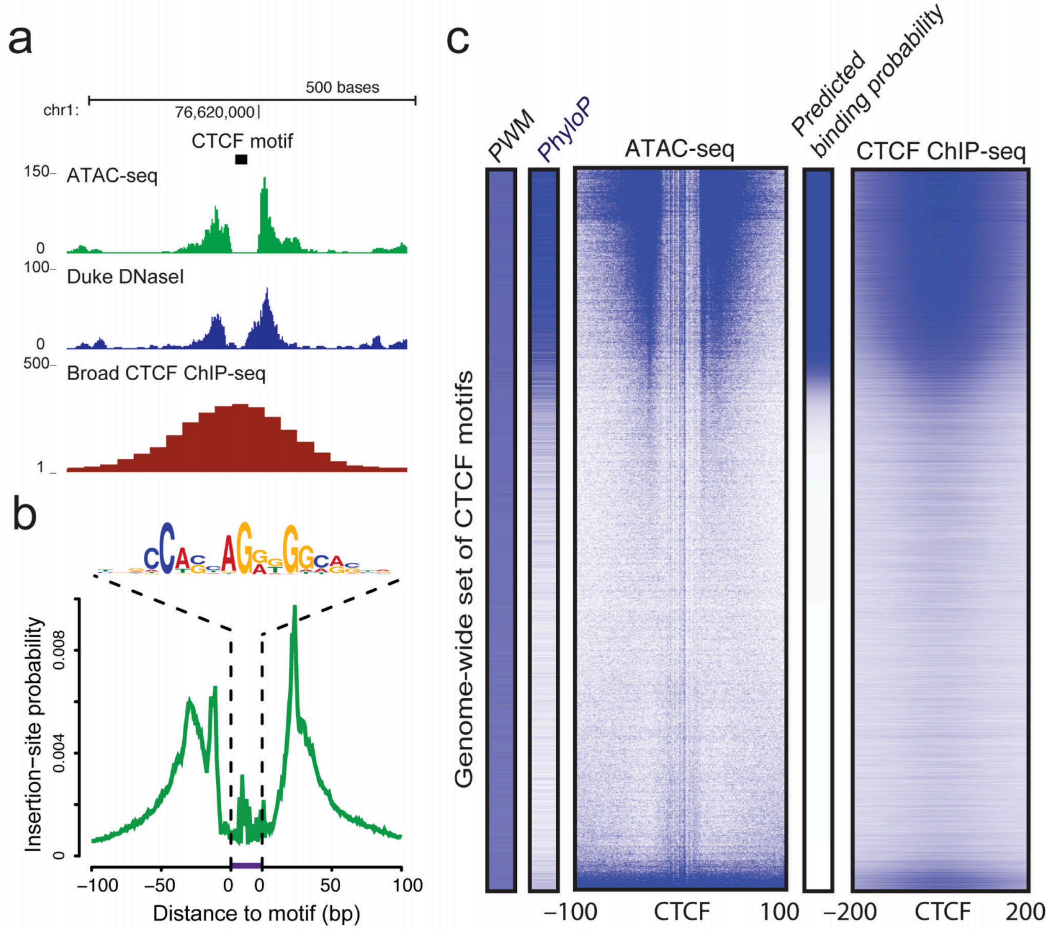


Figure 22.2.: Multimodal chromatin comparisons. From (Buenrostro et al. 2013), Figure 4. (a) CTCF footprints observed in ATAC-seq and DNase-seq data, at a specific locus on chr1. (b) Aggregate ATAC-seq footprint for CTCF (motif shown) generated over binding sites within the genome (c) CTCF predicted binding probability inferred from ATAC-seq data, position weight matrix (PWM) scores for the CTCF motif, and evolutionary conservation (PhyloP). Rightmost column is the CTCF ChIP-seq data (ENCODE) for this GM12878 cell line, demonstrating high concordance with predicted binding probability.

Buenrostro et al. provide a detailed protocol for performing ATAC-Seq and quality control of results (Buenrostro et al. 2015). Updated and modified protocols that improve on signal-to-noise and reduce input DNA requirements have been described.

22.1. Informatics overview

ATAC-Seq protocols typically utilize paired-end sequencing protocols. The reads are aligned to the respective *genome* using `bowtie2`, `BWA`, or other short-read aligner. The result, after appropriate manipulation, often using `samtools`, results in a BAM file. Among other details, the BAM format includes columns for:

```
knitr::include_graphics('imgs/bam_shot.png')
```

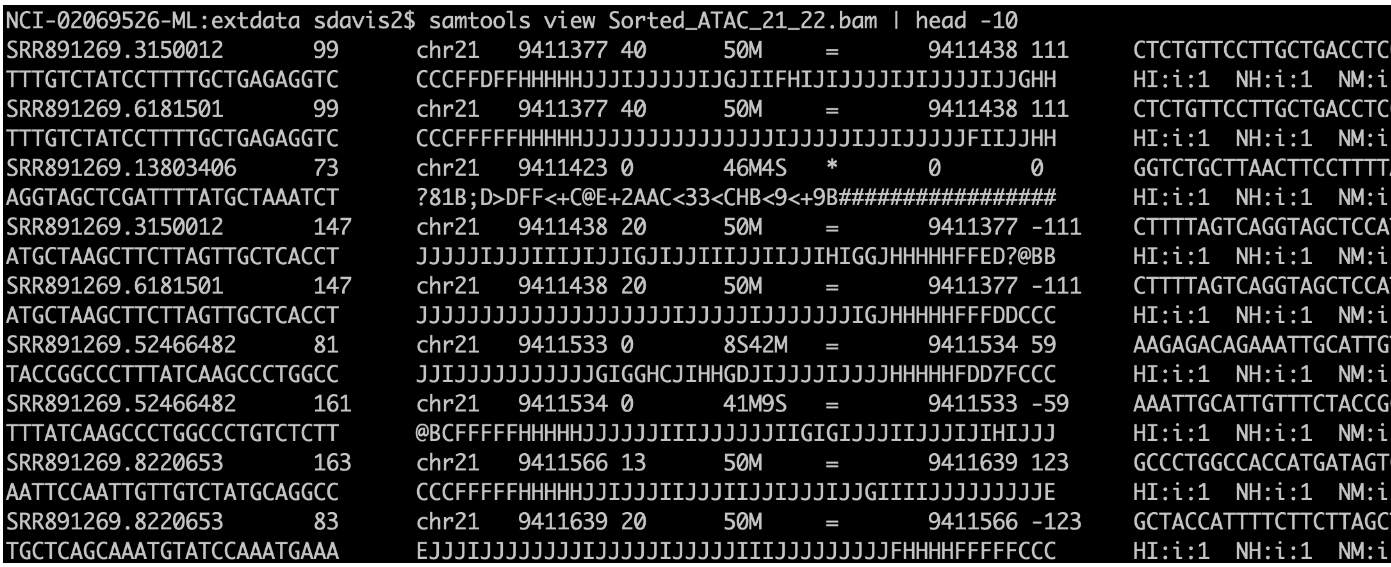


Figure 22.3.: A BAM file in text form. The output of `samtools view` is the text format of the BAM file (called SAM format). Bioconductor and many other tools use BAM files for input. Note that BAM files also often include an index `.bai` file that enables random access into the file; one can read just a genomic region without having to read the entire file.

- sequence name (`chr1`)
- start position (integer)
- a *CIGAR* string that describes the alignment in a compact form
- the sequence to which the pair aligns
- the position to which the pair aligns
- a bit flag field that describes multiple characteristics of the alignment
- the sequence and quality string of the read
- additional tags that tend to be aligner-specific

Duplicate fragments (those with the *same* start and end position of other reads) are marked and likely discarded. Reads that fail to align "properly" are also often excluded from analysis. It is worth noting that most software packages allow simple "marking" of such reads and that there is usually no need to create a special BAM file before proceeding with downstream work.

After alignment and BAM processing, the workflow can switch to *Bioconductor*.

22.2. Working with sequencing data in Bioconductor

The *Bioconductor* project includes several infrastructure packages for dealing with ranges (sequence name, start, end, +/- strand) on sequences (Lawrence et al. 2013) as well as capabilities with working with Fastq files directly (Morgan et al. 2016).

Table 22.1.: Commonly used Bioconductor and their high-level use cases.

Package	Use cases
Rsamtools	low level access to FASTQ, VCF, SAM, BAM, BCF formats
GenomicRanges	Container and methods for handling genomic reagions
GenomicFeatures	Work with transcript databases, gff, gtf and BED formats
GenomicAlignments	Reader for BAM format
rtracklayer	import and export multiple UCSC file formats including BigWig and Bed

As noted in the previous section, the output of an ATAC-Seq experiment is a BAM file. As paired-end sequencing is a commonly-applied approach for ATAC-Seq, the `readGAlignmentPairs` function is the appropriate method to use.

23. Data import and quality control

```
library(GenomicAlignments)
```

Reading a paired-end BAM file looks a bit complicated, but the following code will:

1. Read the included BAM file.
2. Include read pairs only (`isPaired = TRUE`)
3. Include properly paired reads (`isProperPair = TRUE`)
4. Include reads with mapping quality >= 1
5. Add a couple of additional fields, `mapq` (mapping quality) and `isize` (insert size) to the default fields.

```
greenleaf <- readGAlignmentPairs(
    "https://github.com/seandavi/RBiocBook/raw/main/atac-seq/extdata/Sorted_ATAC_21_22.bam"
    param = ScanBamParam(
        mapqFilter = 1,
        flag = scanBamFlag(
            isPaired = TRUE,
            isProperPair = TRUE
        ),
        what = c("mapq", "isize")
    )
)
```

Exercise: What is the class of `greenleaf`? *Exercise*: Use the `GenomicAlignments::first()` accessor to get the first read of the pair as a `GAlignments` object. Save the result as a variable called `gl_first_read`. Use the `mcols` accessor to find the "metadata columns" of `gl_first_read`. *Exercise*: How many read pairs map to each chromosome?

We can make plot of the number of reads mapping to each chromosome.

```r
library(ggplot2)
library(dplyr)
chromCounts <- table(seqnames(greenleaf)) %>%
    data.frame() %>%
    dplyr::rename(chromosome = Var1, count = Freq)
```

To keep things small, the example BAM file includes only chromosomes 21 and 22.

```r
ggplot(chromCounts, aes(x = chromosome, y = count)) +
    geom_bar(stat = "identity") +
    theme(axis.text.x = element_text(angle = 45, hjust = 1))
```

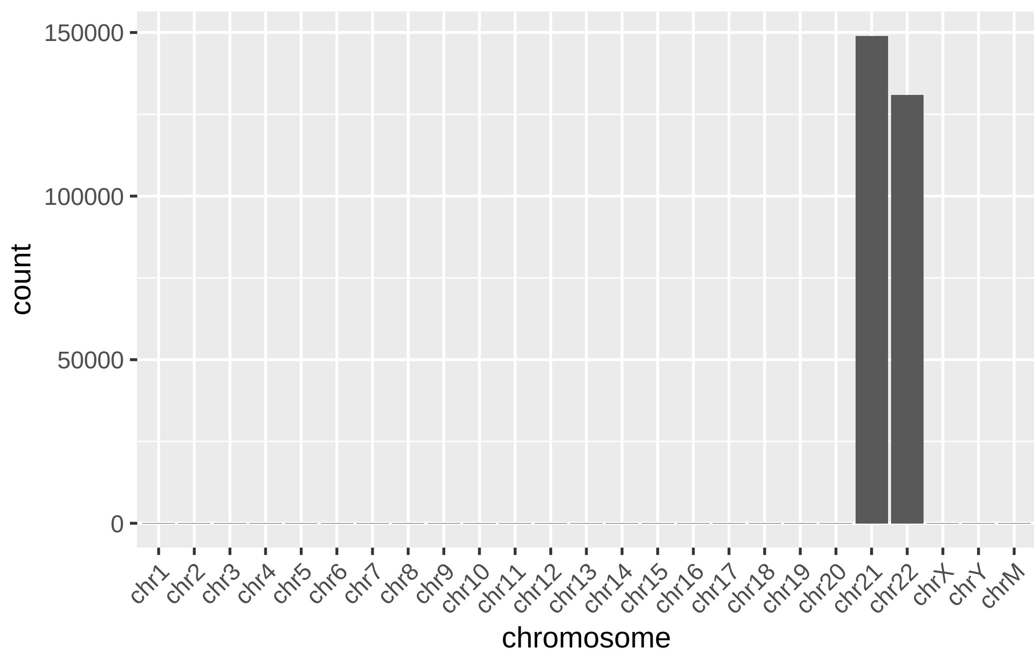


Figure 23.1.: Reads per chromosome. In our example data, we are using only chromosomes 21 and 22.

Normalizing by the chromosome length can yield the reads per megabase which should crudely be similar across all chromosomes.

```r
chromCounts <- chromCounts %>%
    dplyr::mutate(readsPerMb = (count / (seqlengths(greenleaf) / 1e6)))
```

And show a plot. For two chromosomes, this is a little underwhelming.

```
ggplot(chromCounts, aes(x = chromosome, y = readsPerMb)) +
    geom_bar(stat = "identity") +
    theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
    theme_bw()
```

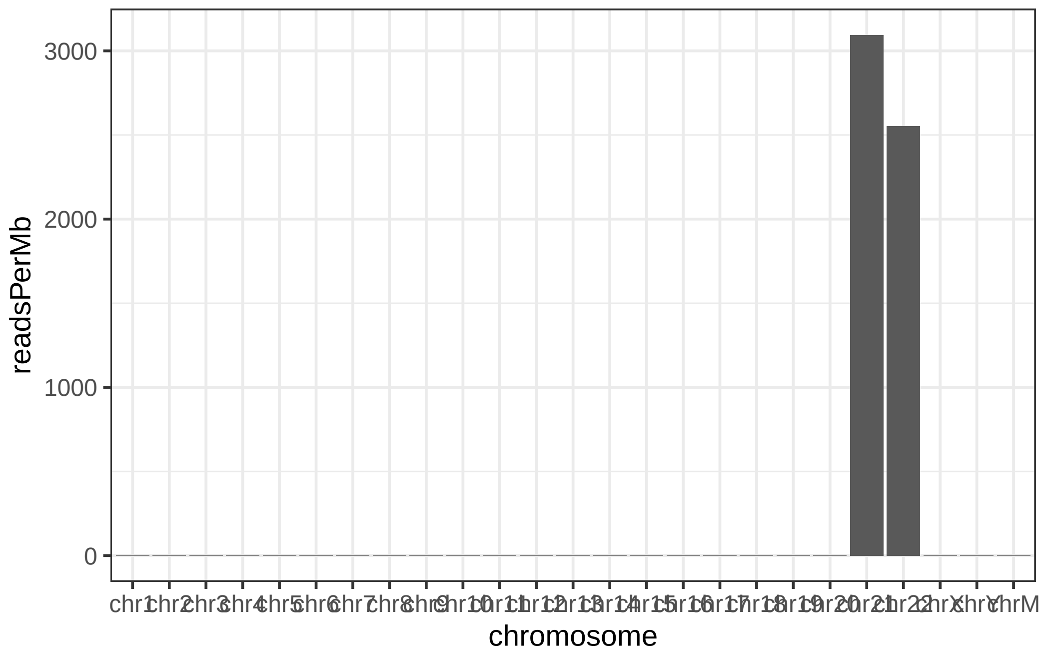


Figure 23.2.: Read counts normalized by chromosome length. This is not a particularly important plot, but it can be useful to see the relative contribution of each chromosome given its length.

23.1. Coverage

The `coverage` method for genomic ranges calculates, for each base, the number of overlapping features. In the case of a BAM file from ATAC-Seq converted to a GAlignmentPairs object, the coverage gives us an idea of the extent to which reads pile up to form peaks.

```
cvg <- coverage(greenleaf)
class(cvg)
```

```
[1] "SimpleRleList"
attr(,"package")
[1] "IRanges"
```

The coverage is returned as a `SimpleRleList` object. Using `names` can get us the names of the elements of the list.

```
names(cvg)
```

```
 [1] "chr1"  "chr2"  "chr3"  "chr4"  "chr5"  "chr6"  "chr7"  "chr8"  "chr9"
[10] "chr10" "chr11" "chr12" "chr13" "chr14" "chr15" "chr16" "chr17" "chr18"
[19] "chr19" "chr20" "chr21" "chr22" "chrX"  "chrY"  "chrM"
```

There is a name for each chromosome. Looking at the `chr21` entry:

```
cvg$chr21
```

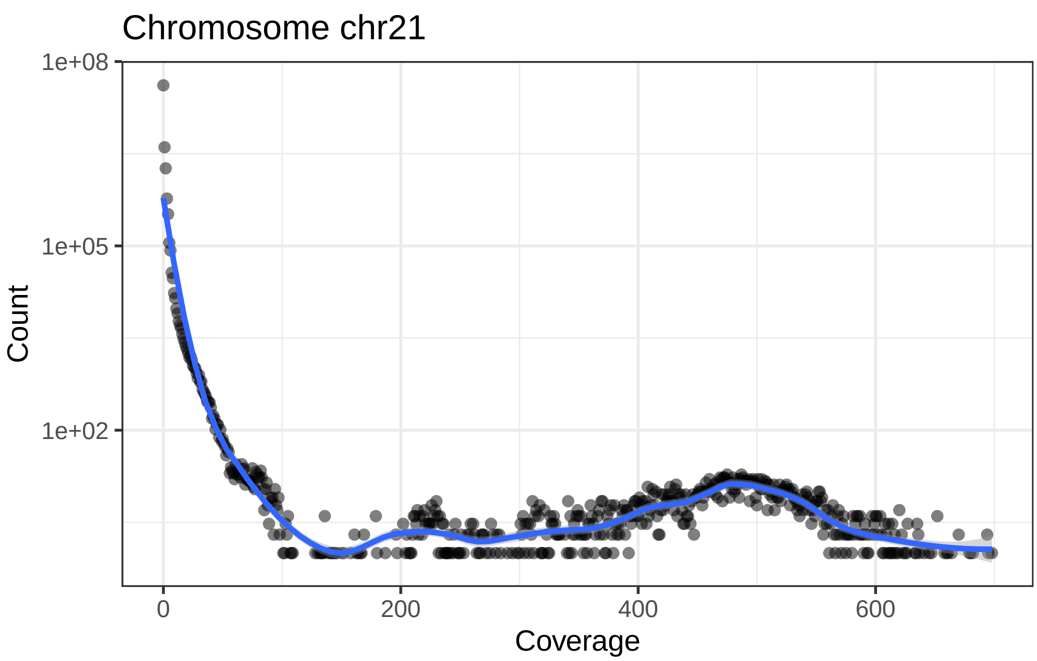
```
integer-Rle of length 48129895 with 397462 runs
  Lengths: 9411376      50      11      50 ...      36      14      28   10806
  Values :       0       2       0       2 ...       1       2       1       0
```

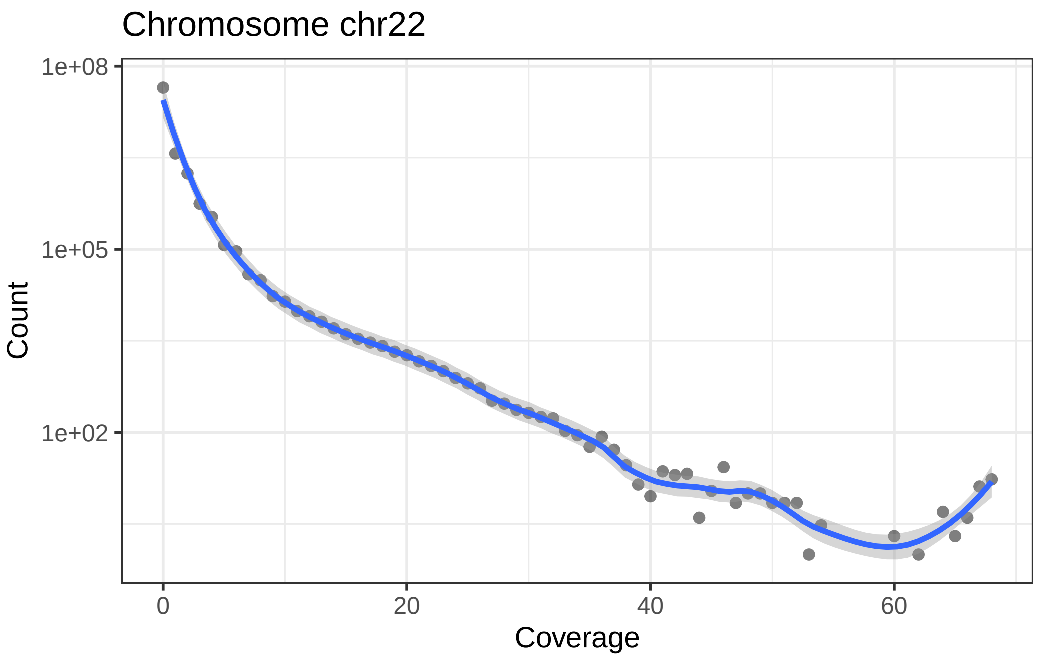
we see that each chromosome is represented as an `Rle`, short for run-length-encoding. Simply put, since along the chromosome there are many repeated values, we can recode the long vector as a set of (length: value) pairs. For example, if the first 9,410,000 base pairs have `0` coverage, we encode that as (9,410,000: 0). Doing that across the chromosome can very significantly reduce the memory use for genomic coverage.

The following little function, `plotCvgHistByChrom` can plot a histogram of the coverage for a chromosome.

```
plotCvgHistByChrom <- function(cvg, chromosome) {
    library(ggplot2)
    cvgcounts <- as.data.frame(table(cvg[[chromosome]]))
    cvgcounts[, 1] <- as.numeric(as.character(cvgcounts[, 1]))
    colnames(cvgcounts) <- c("Coverage", "Count")
    ggplot(cvgcounts, aes(x = Coverage, y = Count)) +
        ggtitle(paste("Chromosome", chromosome)) +
        geom_point(alpha = 0.5) +
        geom_smooth(span = 0.2) +
```

```
        scale_y_log10() +
        theme_bw()
}
for (i in c("chr21", "chr22")) {
    print(plotCvgHistByChrom(cvg, i))
}
```





23.2. Fragment Lengths

The first ATAC-Seq manuscript (Buenrostro et al. 2013) highlighted the relationship between fragment length and nucleosomes (see Figure @ref{fig:flgreenleaf}).

```
knitr::include_graphics("https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3959825/bin/nihms5544
```

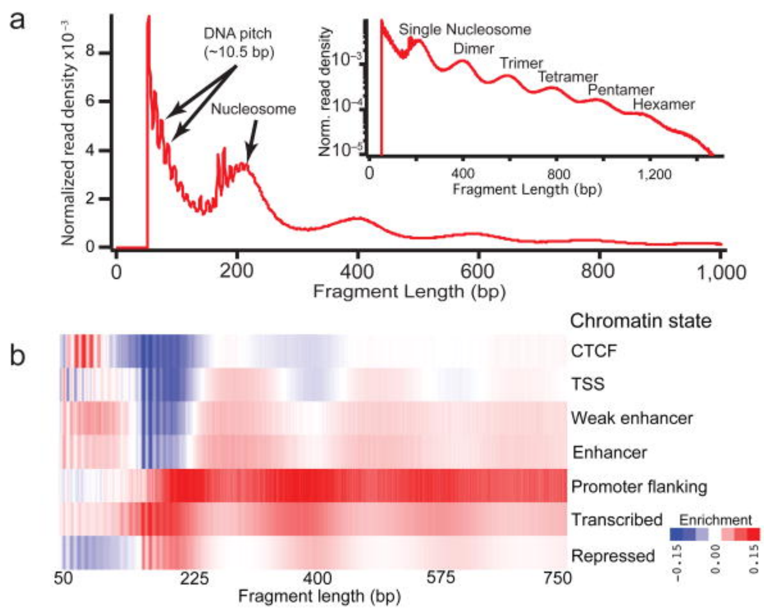


Figure 23.3.: Relationship between fragment length and nucleosome number.

Remember that we loaded the example BAM file with insert sizes (**isize**). We can use that "column" to examine the fragment lengths (another name for insert size). Also, note that the insert size for the **first** read and the **second** are the same (absolute value). Here, we will use **first**.

```
GenomicAlignments::first(greenleaf)
mcols(GenomicAlignments::first(greenleaf))
class(mcols(GenomicAlignments::first(greenleaf)))
head(mcols(GenomicAlignments::first(greenleaf))$isize)
fraglengths <- abs(mcols(GenomicAlignments::first(greenleaf))$isize)
```

We can plot the fragment length density (histogram) using the **density** function.

```
plot(density(fraglengths, bw = 0.05), xlim = c(0, 1000))
```

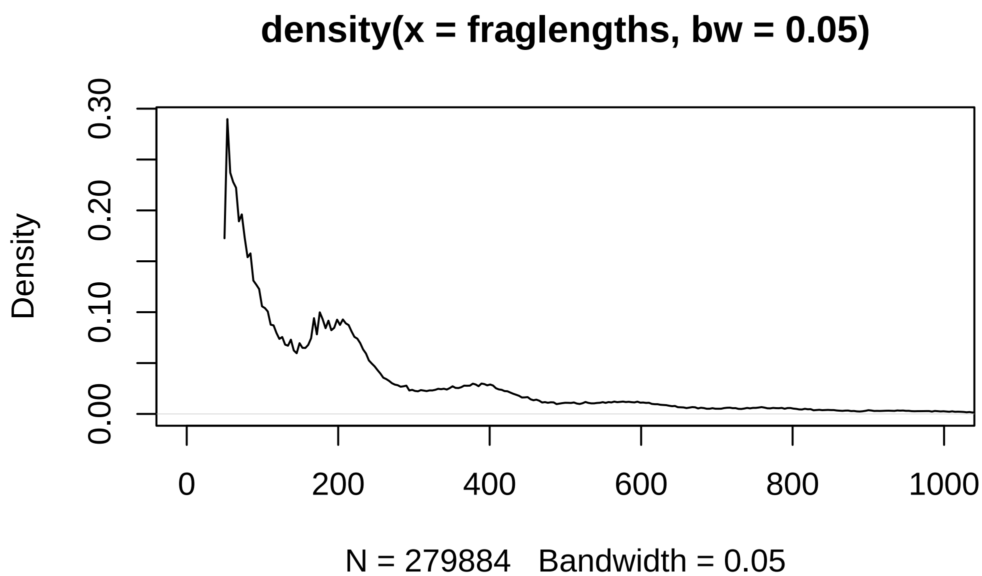
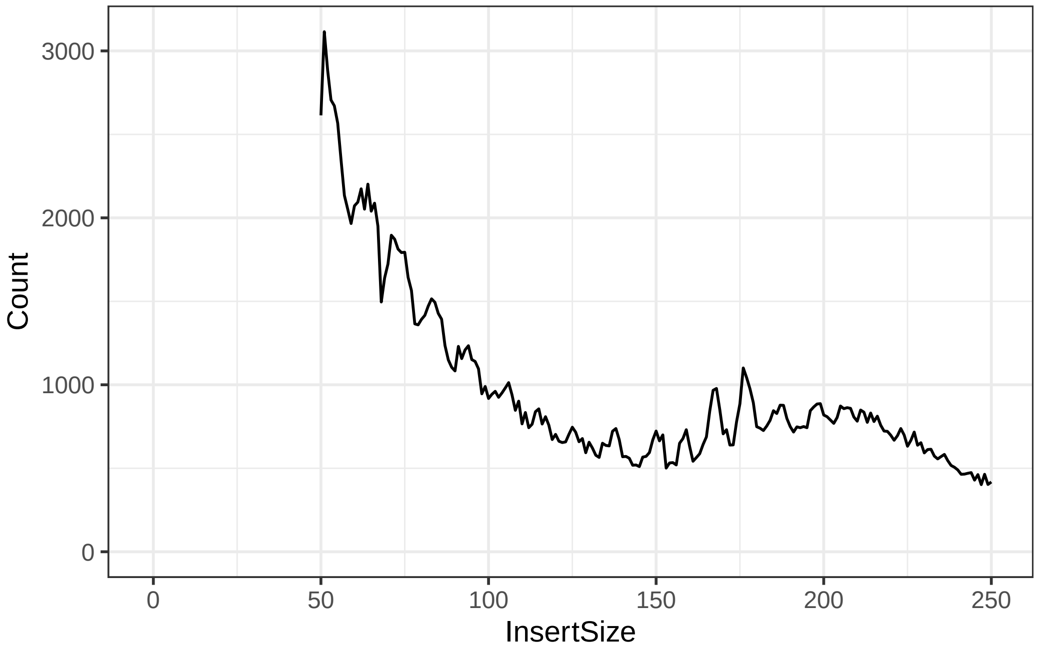


Figure 23.4.: Fragment length histogram.

Exercise: Adjust the `xlim`, `bw`, and try `log="y"` in the plot to highlight features present in figure **??**.

And for fun, the ggplot2 version:

```
library(dplyr)
library(ggplot2)
fragLenPlot <- table(fraglengths) %>%
    data.frame() %>%
    rename(
        InsertSize = fraglengths,
        Count = Freq
    ) %>%
    mutate(
        InsertSize = as.numeric(as.vector(InsertSize)),
        Count = as.numeric(as.vector(Count))
    ) %>%
    ggplot(aes(x = InsertSize, y = Count)) +
    geom_line()
print(fragLenPlot + theme_bw() + lims(x = c(-1, 250)))
```



Knowing that the nucleosome-free regions will have insert sizes shorter than one nucleosome, we can isolate the read pairs that have that characteristic.

```
gl_nf <- greenleaf[mcols(GenomicAlignments::first(greenleaf))$isize < 100]
```

And the mononucleosome reads will be between 187 and 250 base pairs for insert size/fragment length.

```
gl_mn <- greenleaf[mcols(GenomicAlignments::first(greenleaf))$isize > 187 &
    mcols(GenomicAlignments::first(greenleaf))$isize < 250]
```

Finally, we expect nucleosome-free reads to be enriched near the TSS while mononucleosome reads should not be. We will use the *heatmaps* package to take a look at these two sets of reads with respect to the tss of the human genome.

```
library(TxDb.Hsapiens.UCSC.hg19.knownGene)
proms <- promoters(TxDb.Hsapiens.UCSC.hg19.knownGene, 250, 250)
seqs <- c("chr21", "chr22")
seqlevels(proms, pruning.mode = "coarse") <- seqs # only chromosome 21 and 22
```

Take a look at the *heatmaps* package vignette to learn more about the heatmaps package capabilities.

```
library(heatmaps)
gl_nf_hm <- CoverageHeatmap(proms, coverage(gl_nf), coords = c(-250, 250))
label(gl_nf_hm) <- "NucFree"
scale(gl_nf_hm) <- c(0, 10)
plotHeatmapMeta(gl_nf_hm)
```

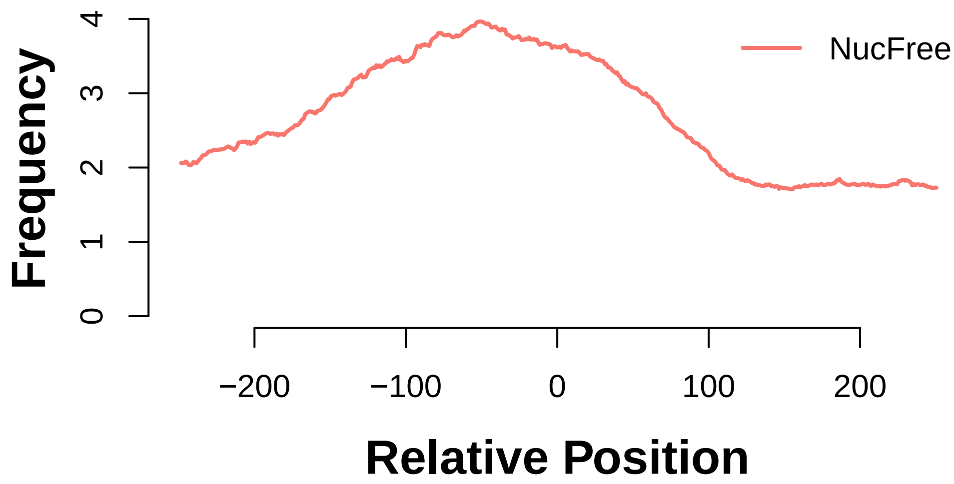


Figure 23.5.: Enrichment of nucleosome free reads just upstream of the TSS.

```
gl_mn_hm <- CoverageHeatmap(proms, coverage(gl_mn), coords = c(-250, 250))
label(gl_mn_hm) <- "MonoNuc"
scale(gl_mn_hm) <- c(0, 10)
plotHeatmapMeta(gl_mn_hm)
```

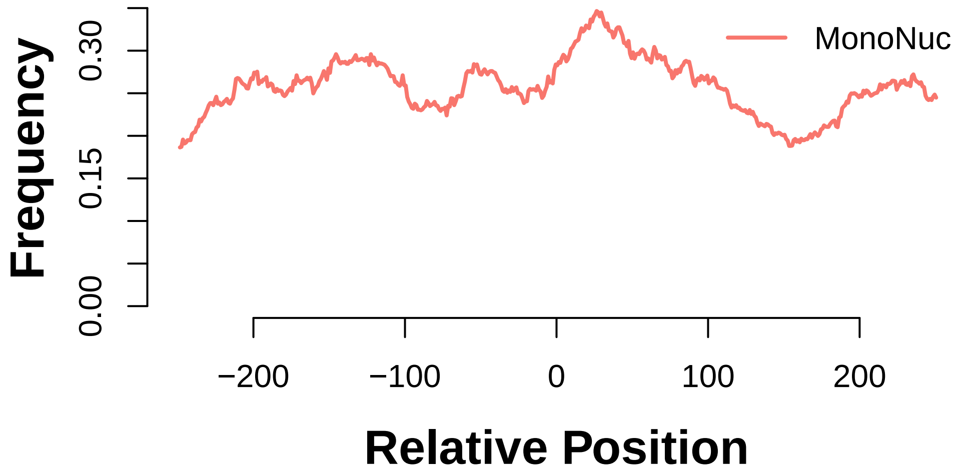


Figure 23.6.: Depletion of nucleosome free reads just upstream of the TSS.

```
plotHeatmapList(list(gl_mn_hm, gl_nf_hm))
```

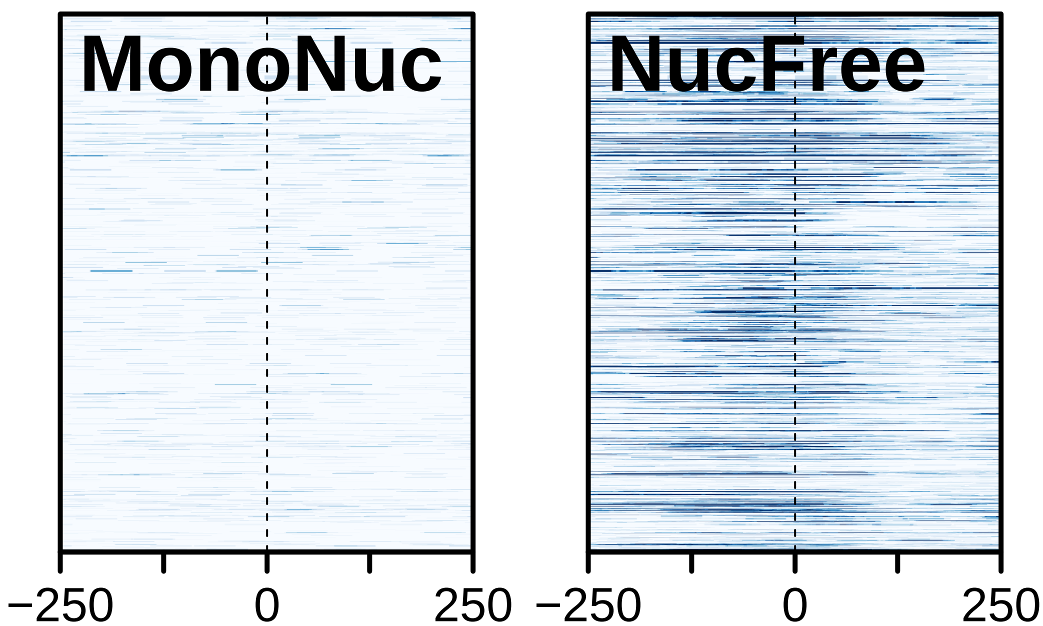


Figure 23.7.: Comparison of signals at TSS. Mononucleosome data on the left, nucleosome-free on the right.

24. Viewing data in IGV

Install IGV from here.

We export the greenleaf data as a BigWig file.

```
library(rtracklayer)
export.bw(coverage(greenleaf), "greenleaf.bw")
```

Exercise: In IGV, choose `hg19`. Then, load the `greenleaf.bw` file and explore chromosomes 21 and 22. *Exercise*: Export the nucleosome-free portion of the data as a BigWig file and examine that in IGV. Where do you expect to see the strongest signals?

25. Additional work

For those working extensively on ATAC-Seq, there is a great workflow/tutorial available from Thomas Carrol:

https://rockefelleruniversity.github.io/RU_ATAC_Workshop.html

Feel free to work through it. In addition to the work above, there is also the *ATACseqQC* package vignette that offers more than just QC. At least a couple more packages are available in *Bioconductor*.

Appendix

Session info

```
R version 4.4.0 (2024-04-24)
Platform: aarch64-apple-darwin20
Running under: macOS Sonoma 14.2.1

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRlapack.dylib;

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: America/New_York
tzcode source: internal

attached base packages:
[1] stats4    stats     graphics  grDevices utils     datasets  methods
[8] base

other attached packages:
 [1] rtracklayer_1.64.0
 [2] heatmaps_1.28.0
 [3] TxDb.Hsapiens.UCSC.hg19.knownGene_3.2.2
 [4] GenomicFeatures_1.56.0
 [5] AnnotationDbi_1.66.0
 [6] dplyr_1.1.4
 [7] ggplot2_3.5.1
 [8] GenomicAlignments_1.40.0
 [9] Rsamtools_2.20.0
[10] Biostrings_2.72.1
[11] XVector_0.44.0
```

```
[12] SummarizedExperiment_1.34.0
[13] Biobase_2.64.0
[14] MatrixGenerics_1.16.0
[15] matrixStats_1.3.0
[16] GenomicRanges_1.56.0
[17] GenomeInfoDb_1.40.1
[18] IRanges_2.38.0
[19] S4Vectors_0.42.0
[20] BiocGenerics_0.50.0
[21] BiocStyle_2.32.0
[22] knitr_1.47

loaded via a namespace (and not attached):
 [1] tidyselect_1.2.1        EBImage_4.46.0          farver_2.1.2
 [4] blob_1.2.4              bitops_1.0-7            fastmap_1.2.0
 [7] RCurl_1.98-1.14         XML_3.99-0.16.1         digest_0.6.35
[10] lifecycle_1.0.4         KEGGREST_1.44.0         RSQLite_2.3.7
[13] magrittr_2.0.3          compiler_4.4.0          rlang_1.1.4
[16] tools_4.4.0             plotrix_3.8-4           utf8_1.2.4
[19] yaml_2.3.8              htmlwidgets_1.6.4       S4Arrays_1.4.1
[22] labeling_0.4.3          bit_4.0.5               curl_5.2.1
[25] DelayedArray_0.30.1     RColorBrewer_1.1-3      KernSmooth_2.23-24
[28] abind_1.4-5             BiocParallel_1.38.0     withr_3.0.0
[31] grid_4.4.0              fansi_1.0.6             colorspace_2.1-0
[34] scales_1.3.0            tinytex_0.51            cli_3.6.2
[37] rmarkdown_2.27          crayon_1.5.2            generics_0.1.3
[40] httr_1.4.7              rjson_0.2.21            DBI_1.2.3
[43] cachem_1.1.0            zlibbioc_1.50.0         splines_4.4.0
[46] parallel_4.4.0          tiff_0.1-12             BiocManager_1.30.23
[49] restfulr_0.0.15         vctrs_0.6.5             Matrix_1.7-0
[52] jsonlite_1.8.8          fftwtools_0.9-11        bit64_4.0.5
[55] jpeg_0.1-10             locfit_1.5-9.9          glue_1.7.0
[58] codetools_0.2-20        gtable_0.3.5            BiocIO_1.14.0
[61] UCSC.utils_1.0.0        munsell_0.5.1           tibble_3.2.1
[64] pillar_1.9.0            htmltools_0.5.8.1       GenomeInfoDbData_1.2.12
[67] R6_2.5.1                evaluate_0.23           lattice_0.22-6
[70] png_0.1-8               memoise_2.0.1           SparseArray_1.4.8
[73] nlme_3.1-165            mgcv_1.9-1              xfun_0.44
[76] pkgconfig_2.0.3
```

MACS2

The MACS2 package is a commonly-used package for calling peaks. Installation and other details are available[1].

```
pip install macs2
```

[1] https://github.com/taoliu/MACS

26. References

27. Transfer Learning in scATAC-seq and scRNA-seq

27.1. Background

Analyzing open chromatin regions has been a crucial aspect of understanding gene regulation and cellular identity. Over the years, several techniques have been developed to identify and study these accessible regions of the genome. One of the earliest methods was DNase-seq, which uses the DNase I enzyme to digest exposed DNA, followed by sequencing of the resulting fragments. This method, introduced in the late 1970s and adapted for high-throughput sequencing in 2006, provided valuable insights into the locations of regulatory elements and transcription factor binding sites. Another technique, called FAIRE-seq (Formaldehyde-Assisted Isolation of Regulatory Elements), was developed in 2007. This method relies on the differential crosslinking of proteins to DNA in open and closed chromatin regions, followed by sequencing of the isolated DNA fragments. FAIRE-seq offered a complementary approach to DNase-seq for identifying open chromatin regions. In 2013, a groundbreaking method called ATAC-seq (Assay for Transposase-Accessible Chromatin using sequencing) was introduced by Buenrostro et al. This technique revolutionized the study of open chromatin by providing a simple, fast, and sensitive approach. ATAC-seq employs a hyperactive Tn5 transposase that simultaneously cuts and inserts adapters into accessible DNA regions. The resulting fragments are then sequenced, revealing the locations of open chromatin. ATAC-seq offers several advantages over previous methods. It requires a small number of cells (as few as 500), making it suitable for studying rare cell types or precious samples. Additionally, the protocol is relatively simple and can be completed in a few hours, compared to the multiple days required for DNase-seq or FAIRE-seq. The high resolution and sensitivity of ATAC-seq have made it a widely adopted technique in the field of epigenomics. The introduction of single-cell ATAC-seq (scATAC-seq) in 2015 further expanded the capabilities of this method. By combining ATAC-seq with microfluidic technologies or combinatorial indexing, researchers can now profile open chromatin landscapes at the single-cell level. This advancement allows for the exploration of cellular heterogeneity, the identification of rare cell types, and the study of dynamic changes in chromatin accessibility during processes like differentiation or disease progression.

27.1.1. Protocol

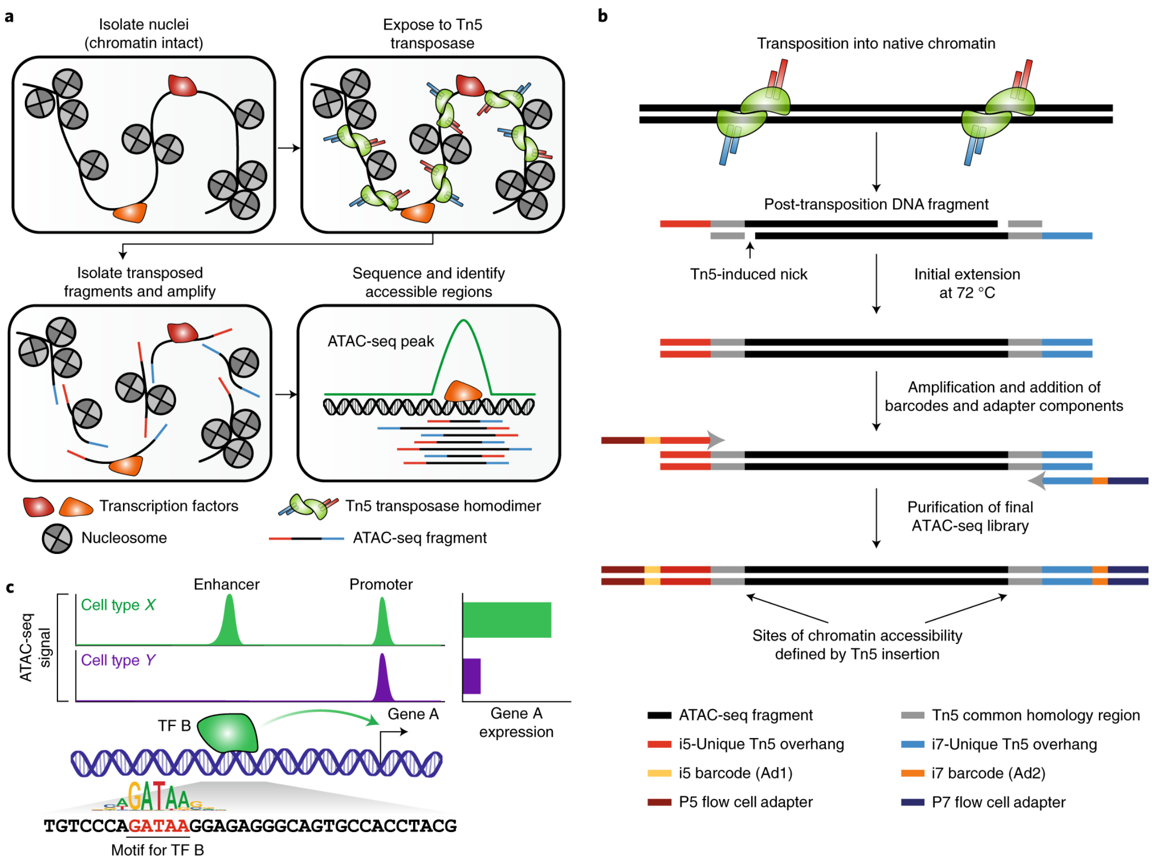


Figure 27.1.

1. Nuclei Isolation and Tn5 Transposition (Figure 27.1 (a))

- **Nuclei Isolation**: The first step involves isolating nuclei from cells while keeping the chromatin intact. This ensures that the native chromatin structure is preserved.
- **Exposure to Tn5 Transposase**: The isolated nuclei are then exposed to Tn5 transposase. The Tn5 enzyme is a hyperactive transposase that simultaneously cuts DNA and inserts sequencing adapters into accessible chromatin regions. This step is crucial as it tags open chromatin areas with sequencing adapters, making them ready for subsequent amplification and sequencing.
- **Fragment Isolation and Amplification**: After transposition, the resulting DNA fragments are isolated. These fragments are then amplified to create a library of

transposed sequences. This library represents the accessible regions of the genome and is ready for sequencing.

- **Sequencing and Identification**: The amplified DNA fragments are sequenced using high-throughput sequencing technologies. The resulting sequences are mapped to the reference genome to identify accessible chromatin regions, known as ATAC-seq peaks. These peaks indicate regions where the chromatin is open and potentially active in gene regulation.

2. **Detailed Mechanism of Tn5 Transposition (Figure 27.1 (b))**

- **Transposition into Native Chromatin**: The Tn5 transposase inserts sequencing adapters into accessible regions of the chromatin. This insertion creates post-transposition DNA fragments, which include the Tn5-induced nick.

- **Initial Extension and Amplification**: Following transposition, the DNA fragments undergo an initial extension at 72°C. This is followed by amplification, during which barcodes and additional adapter components are added. These steps are essential for the preparation of the final ATAC-seq library.

- **Purification and Library Construction**: The amplified fragments are purified to construct the final ATAC-seq library. The sites of chromatin accessibility are defined by the Tn5 insertion, which is marked by specific adapter sequences.

3. **Data Analysis and Interpretation (Figure 27.1 (c))**

- **ATAC-seq Signal and Peaks**: The sequenced data is analyzed to generate an ATAC-seq signal, which shows the read density across the genome. Peaks in the ATAC-seq signal correspond to regions of open chromatin. The example in the figure shows differential chromatin accessibility between two cell types (Cell type X and Cell type Y). Each cell type exhibits unique peaks, indicating distinct regulatory regions.

- **Transcription Factor Binding and Gene Expression**: The open chromatin regions often contain binding sites for transcription factors (TFs). For instance, the motif for a specific TF (TF B) can be identified within a peak. Binding of TF B to its motif within an enhancer or promoter region can regulate the expression of a nearby gene (Gene A). The figure illustrates how the binding of TF B to its motif leads to gene A expression in one cell type but not in another, highlighting the functional impact of chromatin accessibility on gene regulation.

	trimming	read QC	mapping	deduplication	filtering	signal generation	peak calling	QC	downstream analyses
AIAP	cutadapt	FastQC	bwa	picard	samtools methylQA	UCSC tools	MACS2	MultiQC	DESeq2
ATAC2GRN	NA	NA	bowtie2	NA	NA	NA	HOMER	NA	HINT
ATAC-pipe	custom python	custom python	bowtie2	picard	samtools	UCSC tools	MACS2	custom python	CENTIPEDE DESeq2 HOMER
ATACProc	trim_adapters.py*	NA	bowtie2	picard DeepTools	samtools DeepTools	UCSC tools DeepTools	MACS2	ataqv	HINT-ATAC HOMER DeepTools custom python
CIPHER	BBDUK	FastQC	bbmap bowtie2 bwa hisat2 star	NA	samtools	DeepTools	MACS2 epic	MultiQC	NA
ENCODE	trimmomatic cutadapt	NA	bowtie2 bwa	picard	samtools bedtools	UCSC tools	MACS2	custom code	IDR
esATAC	AdapterRemoval	NA	Rbowtie2	custom R	NA	custom R	F-Seq	custom R	ChIPpeakAnno
GUAVA	cutadapt	FastQC	bowtie2	NA	NA	UCSC tools	MACS2	custom code	DESeq2 ChIPpeakAnno
I-ATAC	trimmomatic	FastQC	bwa	picard	NA	NA	MACS2	NA	NA
nfcore/atacseq	Trim Galore!†	FastQC	bwa	picard	samtools bedtools pysam bamtools	bedtools UCSC tools	MACS2	ataqv	DESeq2
PEPATAC	skewer trimmomatic trim_adapters.py‡	FastQC	bowtie2 bwa	samblaster picard samtools	samtools bedtools	custom python	MACS2 F-Seq2 Genrich HMMRATAC HOMER	custom code	HOMER custom code
pyflow-ATAC-seq	atactk§	FastQC	bowtie2	samblaster	samtools	DeepTools	MACS2	ataqv MultiQC	CENTIPEDE
seq2science	Trim Galore!†	FastQC	bowtie2 bwa hisat2 star	picard	samtools	DeepTools	MACS2 Genrich HMMRATAC	MultiQC	custom code
snakePipes ATAC-seq	cutadapt	FastQC	bowtie2	sambamba	samtools	DeepTools	MACS2 Genrich HMMRATAC	MultiQC	CSAW
Tobias Rausch	cutadapt	FastQC	bowtie2	biobambam2	samtools	Alfred	MACS2	Alfred	HOMER custom R tutorial
OVERALL	cutadapt	FastQC	bowtie2	picard	samtools	UCSC tools	MACS2	MultiQC	HOMER DESeq2

Figure 27.2.: ATAC-seq pipelines universally require several common bioinformatic tools. This figure/table shows tools used in various published ATAC-seq pipelines. The figure also displays the typical steps in an ATAC-seq analysis.

27.1.2. Primary data processing

27.1.3. Quality control metrics

In addition to basic read counts and variant quality scores, there are a number of metrics that are valuable for ATAC-seq (or other regional enrichment experiemnts, like ChIP-seq). Figure Figure 27.3 shows example plots from the pepatac workflow.

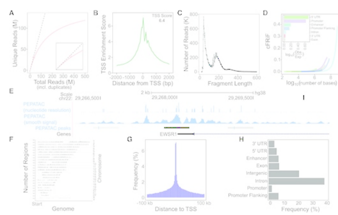


Figure 27.3.: (A) Library complexity plots the read count versus externally calculated deduplicated read counts. Red line is library complexity curve for SRR5427743. Dashed line represents a completely unique library. Red diamond is the externally calculated duplicate read count. (B) TSS enrichment quality control plot. (C) Fragment length distribution showing characteristic peaks at mono-, di-, and tri-nucleosomes. (D) Cumulative fraction of reads in annotated genomic features (cFRiF). Inset: Fraction of reads in those features (FRiF). (E) Signal tracks including: nucleotide-resolution and smoothed signal tracks. PEPATAC default peaks are called using the default pipeline settings for MACS2 (32). (F) Distribution of peaks over the genome. (G) Distribution of peaks relative to TSS. (H) Distribution of peaks in annotated genomic partitions. Data from SRR5427743.

27.2. ATAC-seq and RNA-seq integration

Single-cell transcriptomics has revolutionized our ability to characterize cell states, but a deeper biological understanding requires more than just clustering cells. As new methods emerge to measure different cellular modalities, integrating these datasets becomes a key challenge in better understanding cellular identity and function. For instance, when performing scRNA-seq and scATAC-seq experiments on the same biological system, consistently annotating both datasets with the same cell type labels can be difficult due to the sparsity of scATAC-seq data and the lack of interpretable gene markers in scRNA-seq data.

In a 2019 paper by Stuart, Butler, and colleagues, methods were introduced to integrate scRNA-seq and scATAC-seq datasets from the same biological system. This vignette demonstrates these methods, including:

Using an annotated scRNA-seq dataset to label cells from an scATAC-seq experiment Co-visualizing and co-embedding cells from scRNA-seq and scATAC-seq Projecting scATAC-seq cells onto a UMAP derived from an scRNA-seq experiment

The Signac package, recently developed for analyzing single-cell resolution chromatin datasets like scATAC-seq, is extensively used in this vignette.

The methods are demonstrated using a publicly available ~12,000 human PBMC 'multiome' dataset from 10x Genomics, where scRNA-seq and scATAC-seq profiles were simultaneously collected from the same cells. For the purpose of this vignette, the datasets are treated as if they originated from two different experiments and are integrated together. Since they were originally measured in the same cells, this provides a ground truth for assessing the accuracy of the integration. It is emphasized that the use of the multiome dataset here is for demonstration and evaluation purposes, and users should apply these methods to separately collected scRNA-seq and scATAC-seq datasets.

27.2.1. Setup

```
BiocManager::install('satijalab/seurat-data')
```

The following code loads pre-packaged data from the PBMC Multiome dataset from 10x Genomics.

```
library(SeuratData)
# install the dataset and load requirements
InstallData("pbmcMultiome")
```

We'll be using some additional packages. If you get errors here that a package is not available, you can use `BiocManager::install` to install the missing package and then rerun this step.

```
library(Seurat)
library(Signac)
library(EnsDb.Hsapiens.v86)
library(ggplot2)
library(cowplot)
```

Here, we just load the pre-compiled data. However, if you have your own data, you'd load these data using special data importers or by reading the parts of your data separately.

```
# load both modalities
pbmc.rna <- LoadData("pbmcMultiome", "pbmc.rna")
pbmc.atac <- LoadData("pbmcMultiome", "pbmc.atac")
```

(These next details are taken directly from the Seurat vignette, so I'm going to just blindly follow them.)

```
pbmc.rna[["RNA"]] <- as(pbmc.rna[["RNA"]], Class = "Assay5")
# repeat QC steps performed in the WNN vignette
pbmc.rna <- subset(pbmc.rna, seurat_annotations != "filtered")
pbmc.atac <- subset(pbmc.atac, seurat_annotations != "filtered")
```

27.2.2. RNA-seq processing

This section just follows the Seurat RNA-seq pipeline. At a high level, the steps include:

1. **Normalization**: This line normalizes the RNA data. Normalization typically adjusts the expression measurements to account for differences in sequencing depth or other technical variations across cells. In Seurat, the `NormalizeData` function scales the gene expression measurements for each cell by the total expression, multiplies by a scaling factor (default is 10,000), and log-transforms the result.
2. **Finding Variable Features**: This step identifies the genes that show high variability across cells. These highly variable genes are more likely to capture the biological differences between cells. The `FindVariableFeatures` function selects these genes for downstream analysis.
3. **Scaling the Data**: This line scales the data to have a mean of zero and a variance of one. This standardization step is important for downstream dimensionality reduction techniques like PCA (Principal Component Analysis). The `ScaleData` function centers and scales the data.
4. **Running Principal Component Analysis (PCA)**: PCA is a dimensionality reduction technique that reduces the data to a set of principal components (PCs). These PCs capture the most significant sources of variation in the data. The `RunPCA` function in Seurat performs PCA and stores the results in the object.

5. **Running Uniform Manifold Approximation and Projection (UMAP)**: UMAP is another dimensionality reduction technique that is often used for visualization of high-dimensional data. It captures the local and global structure of the data more effectively than PCA for certain types of data. The `RunUMAP` function runs UMAP on the RNA data, using the first 30 principal components (as specified by `dims = 1:30`).

```
# Perform standard analysis of each modality independently RNA analysis
pbmc.rna <- NormalizeData(pbmc.rna)
pbmc.rna <- FindVariableFeatures(pbmc.rna)
pbmc.rna <- ScaleData(pbmc.rna)
pbmc.rna <- RunPCA(pbmc.rna)
pbmc.rna <- RunUMAP(pbmc.rna, dims = 1:30)
```

27.2.3. Annotate ATAC-seq regions

```
# ATAC analysis add gene annotation information
annotations <- GetGRangesFromEnsDb(ensdb = EnsDb.Hsapiens.v86)
seqlevelsStyle(annotations) <- "UCSC"
genome(annotations) <- "hg38"
Annotation(pbmc.atac) <- annotations
```

And take a look at what we added:

```
head(Annotation(pbmc.atac))
```

```
GRanges object with 6 ranges and 5 metadata columns:
                  seqnames        ranges strand |           tx_id   gene_name
                     <Rle>     <IRanges>  <Rle> |     <character> <character>
  ENSE00001489430     chrX 276322-276394      + | ENST00000399012      PLCXD1
  ENSE00001536003     chrX 276324-276394      + | ENST00000484611      PLCXD1
  ENSE00002160563     chrX 276353-276394      + | ENST00000430923      PLCXD1
  ENSE00001750899     chrX 281055-281121      + | ENST00000445062      PLCXD1
  ENSE00001489388     chrX 281192-281684      + | ENST00000381657      PLCXD1
  ENSE00001719251     chrX 281194-281256      + | ENST00000429181      PLCXD1
                          gene_id   gene_biotype     type
                      <character>    <character> <factor>
  ENSE00001489430 ENSG00000182378 protein_coding     exon
```

```
ENSE00001536003 ENSG00000182378 protein_coding     exon
ENSE00002160563 ENSG00000182378 protein_coding     exon
ENSE00001750899 ENSG00000182378 protein_coding     exon
ENSE00001489388 ENSG00000182378 protein_coding     exon
ENSE00001719251 ENSG00000182378 protein_coding     exon
-------
seqinfo: 25 sequences (1 circular) from hg38 genome
```

27.2.4. ATAC-seq processing

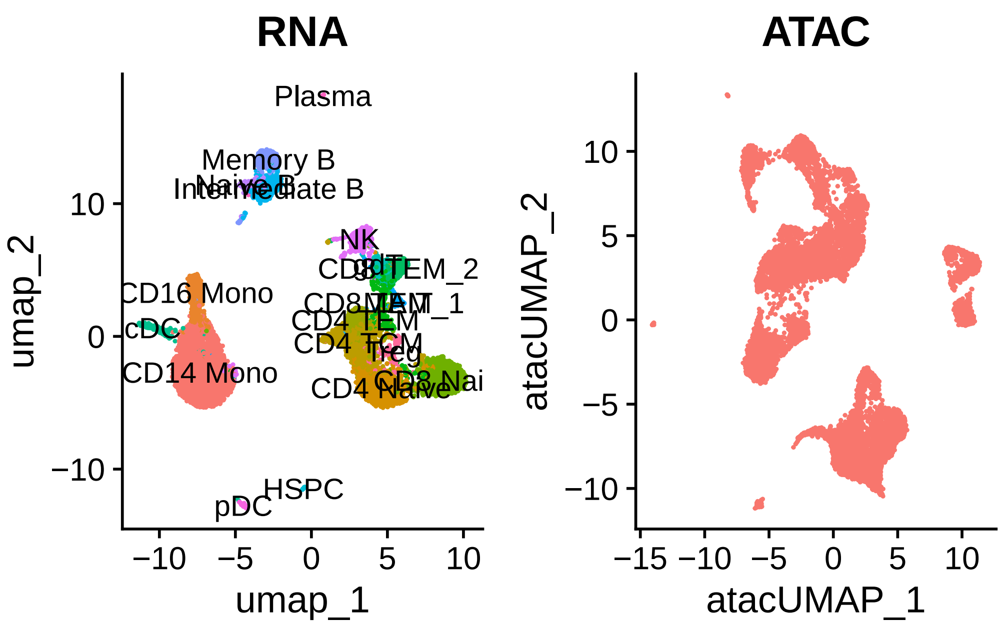
- **Normalization** Signac performs term frequency-inverse document frequency (TF-IDF) normalization. This is a two-step normalization procedure, that both normalizes across cells to correct for differences in cellular sequencing depth, and across peaks to give higher values to more rare peaks.
- **Feature selection** The low dynamic range of scATAC-seq data makes it challenging to perform variable feature selection, as we do for scRNA-seq. Instead, we can choose to use only the top n% of features (peaks) for dimensional reduction, or remove features present in less than n cells with the FindTopFeatures() function. Here we will use all features, though we have seen very similar results when using only a subset of features (try setting min.cutoff to 'q75' to use the top 25% all peaks), with faster runtimes. Features used for dimensional reduction are automatically set as VariableFeatures() for the Seurat object by this function.
- **Dimension reduction** We next run singular value decomposition (SVD) on the TD-IDF matrix, using the features (peaks) selected above. This returns a reduced dimension representation of the object (for users who are more familiar with scRNA-seq, you can think of this as analogous to the output of PCA).

The process described below for dimensionality reduction combining Term Frequency-Inverse Document Frequency (TFIDF) and Singular Value Decomposition (SVD) is called Latent Semantic Indexing (LSI) and was first described here. Suffice it so say that since our ATAC-seq data are very "sparse

```r
# We exclude the first dimension as this is typically correlated with sequencing depth
pbmc.atac <- RunTFIDF(pbmc.atac)
pbmc.atac <- FindTopFeatures(pbmc.atac, min.cutoff = "q0")
pbmc.atac <- RunSVD(pbmc.atac)
pbmc.atac <- RunUMAP(pbmc.atac, reduction = "lsi", dims = 2:30, reduction.name = "umap.atac
```

Now, plot the results.

```
p1 <- DimPlot(pbmc.rna, group.by = "seurat_annotations", label = TRUE) + NoLegend() + ggtit
p2 <- DimPlot(pbmc.atac, group.by = "orig.ident", label = FALSE) + NoLegend() + ggtitle("AT
p1 + p2
```



The UMAP visualization reveals the presence of multiple cell groups in human blood. If you are familiar with scRNA-seq analyses of PBMC, you may recognize the presence of certain myeloid and lymphoid populations in the scATAC-seq data. However, annotating and interpreting clusters is more challenging in scATAC-seq data as much less is known about the functional roles of noncoding genomic regions than is known about protein coding regions (genes).

We can try to quantify the activity of each gene in the genome by assessing the chromatin accessibility associated with the gene, and create a new gene activity assay derived from the scATAC-seq data. Here we will use a simple approach of summing the fragments intersecting the gene body and promoter region (we also recommend exploring the Cicero tool, which can accomplish a similar goal, and we provide a vignette showing how to run Cicero within a Signac workflow here).

To create a gene activity matrix, we extract gene coordinates and extend them to include the 2 kb upstream region (as promoter accessibility is often correlated with gene expression). We then count the number of fragments for each cell that map to each of these regions, using the using the FeatureMatrix() function. These steps are automatically performed by the GeneActivity() function:

```
# quantify gene activity
gene.activities <- GeneActivity(pbmc.atac, features = VariableFeatures(pbmc.rna))

# add gene activities as a new assay
pbmc.atac[["ACTIVITY"]] <- CreateAssayObject(counts = gene.activities)
```

```
# normalize gene activities
DefaultAssay(pbmc.atac) <- "ACTIVITY"
pbmc.atac <- NormalizeData(pbmc.atac)
pbmc.atac <- ScaleData(pbmc.atac, features = rownames(pbmc.atac))
```

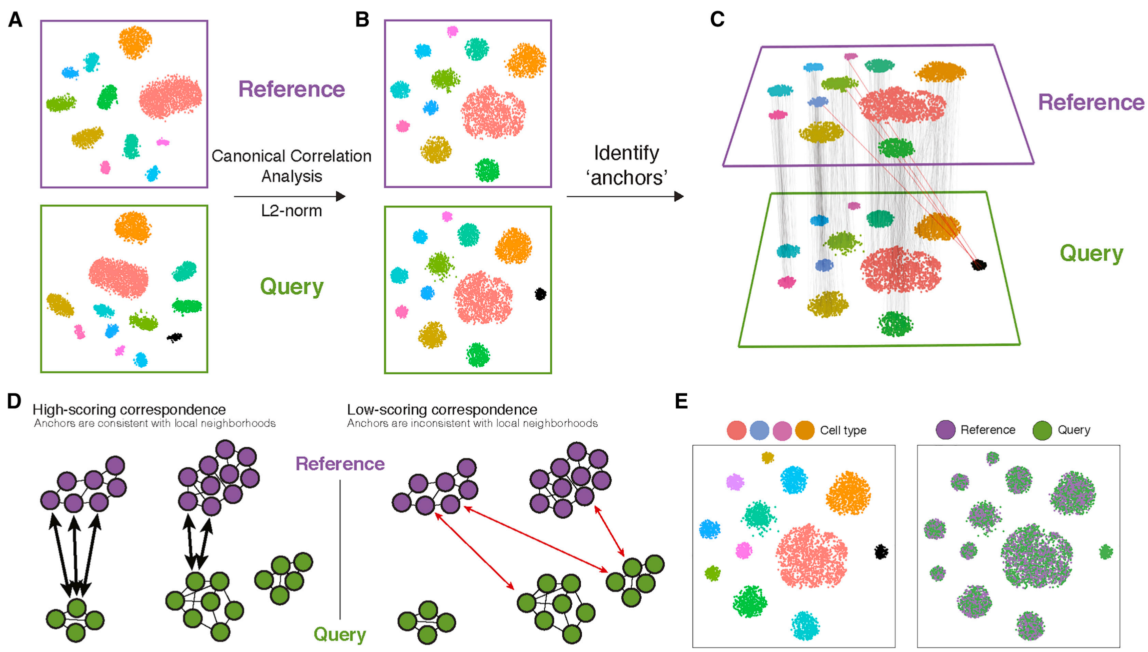


Figure 27.4.

To map cell identities from RNA-seq to ATAC-seq, we follow the steps outlined in the paper by Stuart et al.

In Figure 27.4, (A) Representation of two datasets, reference and query, each of which originates from a separate single-cell experiment. The two datasets share cells from similar biological states, but the query dataset contains a unique population (in black). (B) We perform canonical correlation analysis, followed by L2 normalization of the canonical correlation vectors, to project the datasets into a subspace defined by shared correlation structure across datasets. (C) In the shared space, we identify pairs of MNNs across reference and query cells. These should represent cells in a shared biological state across datasets (gray lines) and serve as anchors to guide dataset integration. In principle, cells in unique populations should not participate in anchors, but in practice, we observe "incorrect" anchors at low frequency (red lines). (D) For each anchor pair, we assign a score based on the consistency of anchors across the neighborhood structure of each dataset. (E) We utilize anchors and their scores to compute "correction" vectors for each query cell, transforming its expression so it can be jointly analyzed as part of an integrated reference.

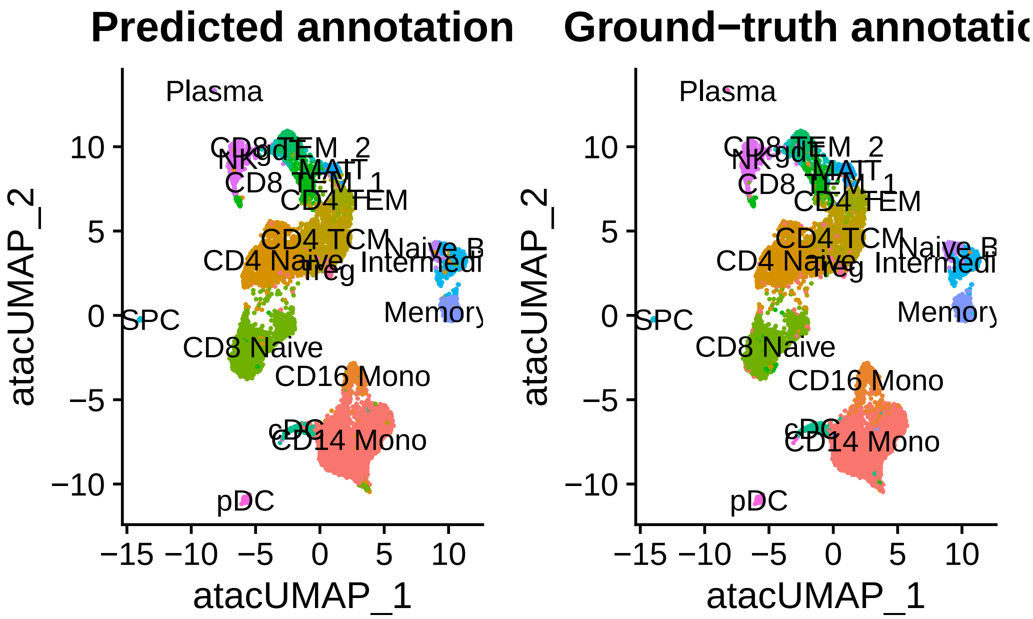
```
# Identify anchors
transfer.anchors <- FindTransferAnchors(reference = pbmc.rna, query = pbmc.atac, features =
    reference.assay = "RNA", query.assay = "ACTIVITY", reduction = "cca")
```

After identifying anchors, we can transfer annotations from the scRNA-seq dataset onto the scATAC-seq cells. The annotations are stored in the seurat_annotations field, and are provided as input to the refdata parameter. The output will contain a matrix with predictions and confidence scores for each ATAC-seq cell.

```
celltype.predictions <- TransferData(anchorset = transfer.anchors, refdata = pbmc.rna$seura
    weight.reduction = pbmc.atac[["lsi"]], dims = 2:30)
pbmc.atac <- AddMetaData(pbmc.atac, metadata = celltype.predictions)
```

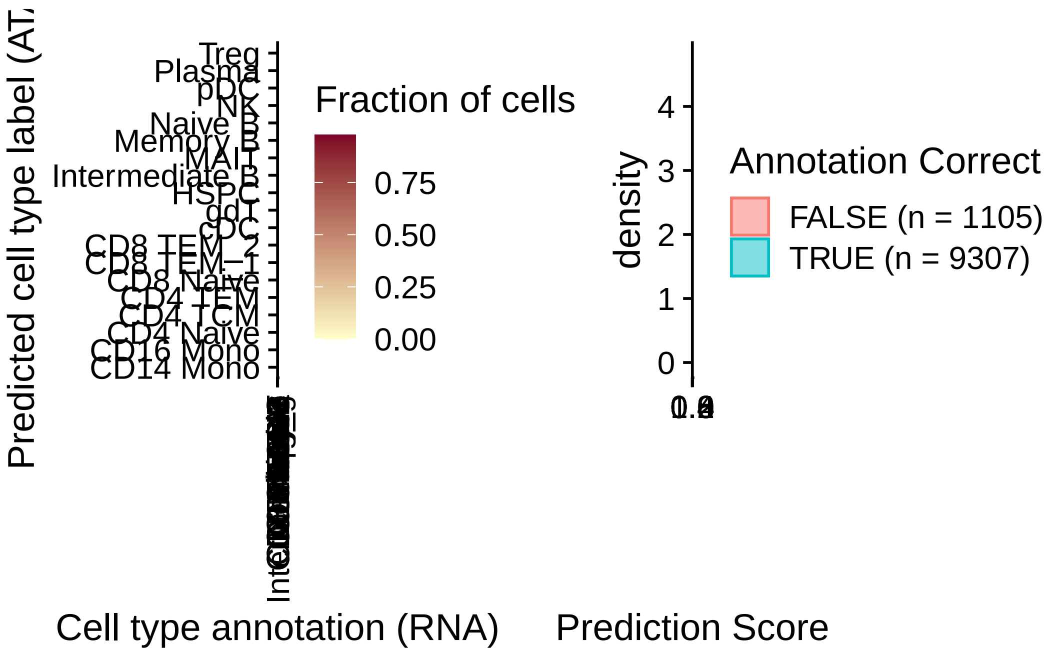
After performing transfer, the ATAC-seq cells have predicted annotations (transferred from the scRNA-seq dataset) stored in the predicted.id field. Since these cells were measured with the multiome kit, we also have a ground-truth annotation that can be used for evaluation. You can see that the predicted and actual annotations are extremely similar.

```
pbmc.atac$annotation_correct <- pbmc.atac$predicted.id == pbmc.atac$seurat_annotations
p1 <- DimPlot(pbmc.atac, group.by = "predicted.id", label = TRUE) + NoLegend() + ggtitle("P
p2 <- DimPlot(pbmc.atac, group.by = "seurat_annotations", label = TRUE) + NoLegend() + ggti
p1 | p2
```



In this example, the annotation for an scATAC-seq profile is correctly predicted via scRNA-seq integration ~90% of the time. In addition, the prediction.score.max field quantifies the uncertainty associated with our predicted annotations. We can see that cells that are correctly annotated are typically associated with high prediction scores (>90%), while cells that are incorrectly annotated are associated with sharply lower prediction scores (<50%). Incorrect assignments also tend to reflect closely related cell types (i.e. Intermediate vs. Naive B cells).

```
predictions <- table(pbmc.atac$seurat_annotations, pbmc.atac$predicted.id)
predictions <- predictions/rowSums(predictions)  # normalize for number of cells in each ce
predictions <- as.data.frame(predictions)
p1 <- ggplot(predictions, aes(Var1, Var2, fill = Freq)) + geom_tile() + scale_fill_gradient
    low = "#ffffc8", high = "#7d0025") + xlab("Cell type annotation (RNA)") + ylab("Predict
    theme_cowplot() + theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust = 1))

correct <- length(which(pbmc.atac$seurat_annotations == pbmc.atac$predicted.id))
incorrect <- length(which(pbmc.atac$seurat_annotations != pbmc.atac$predicted.id))
data <- FetchData(pbmc.atac, vars = c("prediction.score.max", "annotation_correct"))
p2 <- ggplot(data, aes(prediction.score.max, fill = annotation_correct, colour = annotation
    geom_density(alpha = 0.5) + theme_cowplot() + scale_fill_discrete(name = "Annotation Co
    labels = c(paste0("FALSE (n = ", incorrect, ")"), paste0("TRUE (n = ", correct, ")")))
    labels = c(paste0("FALSE (n = ", incorrect, ")"), paste0("TRUE (n = ", correct, ")")))
p1 + p2
```



27.3. Transfer learning

In this demonstration, we will explore the concept of transfer learning using Principal Component Analysis (PCA). Transfer learning allows us to leverage knowledge gained from one dataset and apply it to another related dataset. We will showcase this by dividing a dataset into two pieces and projecting the second dataset into the principal components derived from the first dataset.

27.3.1. Loading the Data

First, let's load the required libraries:

```r
library(GEOquery)
library(SummarizedExperiment)
```

We will use the `GEOquery` package to retrieve a dataset from the Gene Expression Omnibus (GEO) database and convert it into a `SummarizedExperiment` object:

```
se = as(getGEO("GSE103512")[[1]], "SummarizedExperiment")
```

27.3.2. Selecting the Most Variable Genes

To focus on the most informative genes, we will select the top 250 most variable genes based on their standard deviation. Let's denote the expression matrix as X, where rows represent genes and columns represent samples.

```
# get the top 250 most variable genes
variable_rows = order(apply(assays(se)$exprs, 1, sd), decreasing = TRUE)[1:250]
```

We subset the `SummarizedExperiment` object to include only the selected genes:

```
se_subset <- se[variable_rows,]
```

27.3.3. Splitting the Dataset

Now, we will split the dataset into two pieces, simulating the collection of two **separate** datasets with the same genes. This will allow us to demonstrate transfer learning. Let's denote the subsets as X_1 and X_2.

```
split_vector = sample(c(TRUE,FALSE), ncol(se_subset), replace=TRUE)
se_subset_1 = se_subset[,split_vector]
se_subset_2 = se_subset[,!split_vector]
```

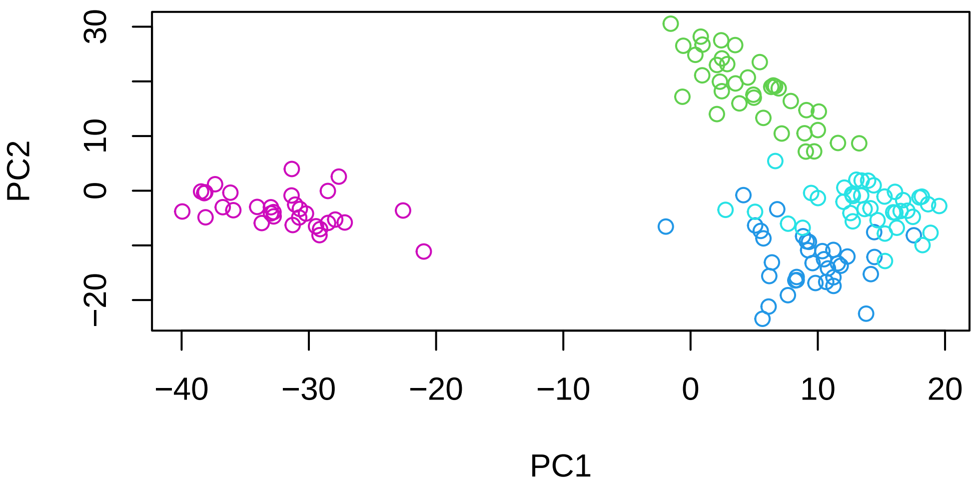
27.3.4. Performing PCA on the First Subset

We perform PCA on the first subset (X_1) to obtain the principal components. PCA seeks to find a set of orthogonal vectors (principal components) that capture the maximum variance in the data. The principal components are the eigenvectors of the covariance matrix of X_1.

```
pc_subset1 = prcomp(t(assays(se_subset_1)$exprs))
```

Let's visualize the samples in the principal component space, colored by their cancer type:

```
plot(pc_subset1$x, col=as.numeric(as.factor(se_subset_1$cancer.type.ch1))+2)
```



27.3.5. Projecting the Second Subset

Now, let's use the PCA model trained on X_1 to project the samples from X_2 into the same principal component space. This is where transfer learning comes into play. We can represent the projection matrix as P, which consists of the top principal components from X_1.

```
pred_subset2 <- predict(pc_subset1,t(assay(se_subset_2,'exprs')))
```

Mathematically, the projection of X_2 into the principal component space is given by:

$X_2^{(p)} = X_2 \cdot P$

where $X_2^{(p)}$ represents the projected samples from X_2 in the principal component space.

In PCA, the principal components represent a new coordinate system that is aligned with the directions of maximum variance in the data. The process of finding these principal components can be thought of as a rotation of the original coordinate system. Consider the original feature space, where each dimension corresponds to a variable (gene in our example). The data points (samples) are scattered in this high-dimensional space. PCA identifies the directions in which the data varies the most, and these directions become the principal components. Geometrically, the principal components form a new orthogonal coordinate system. The first principal component (PC1) aligns with the direction of maximum variance, the second principal component (PC2) aligns with the direction of the second-highest variance (orthogonal to PC1), and so on. When we perform PCA on the first subset (X_1), we obtain the principal components P. These principal components define the rotation matrix that transforms the original coordinate system to the new PCA coordinate system. Now, let's consider the "predict" process, where we project the samples from the second subset (X_2) into the principal component space derived from X_1. Geometrically, this can be understood as follows:

The samples from X_2 are originally represented in the same high-dimensional feature space as X_1. By using the "predict" function with the PCA model trained on X_1, we are essentially applying the rotation matrix P to the samples from X_2. The rotation matrix P transforms the coordinates of the samples from X_2 into the new PCA coordinate system defined by the principal components of X_1. In the PCA coordinate system, the samples from X_2 are represented by their projections onto the principal components.

Mathematically, the projection of X_2 onto the principal component space is given by: $X_2^{(p)} = X_2 \cdot P$ where $X_2^{(p)}$ represents the projected samples from X_2 in the principal component space. Geometrically, this projection can be visualized as follows:

Each sample from X_2 is represented as a point in the original high-dimensional feature space. The rotation matrix P defines the new PCA coordinate system, where the axes are the principal components. The "predict" process maps each sample from X_2 onto the new PCA coordinate system by applying the rotation defined by P. The projected samples $X_2^{(p)}$ represent the coordinates of the samples from X_2 in the PCA coordinate system.

By projecting the samples from X_2 into the PCA space derived from X_1, we can analyze how well the structure and variability of X_2 align with the principal components learned from X_1. If the projected samples from X_2 exhibit similar patterns or groupings as the samples from X_1 in the PCA space, it indicates that the knowledge learned from X_1 effectively captures the underlying structure of X_2.

The "predict" process in PCA can be understood as a rotation of the original coordinate system to align with the directions of maximum variance, followed by a projection of new samples onto the rotated coordinate system defined by the principal components.

27.3.6. Comparing the Subsets in the Principal Component Space

Finally, we can compare the distribution of samples from both subsets in the principal component space:

```
par(mfrow=c(1,2))
plot(pc_subset1$x, col=as.numeric(as.factor(se_subset_1$cancer.type.ch1))+2)
plot(pred_subset2[,1], pred_subset2[,2], col=as.numeric(as.factor(se_subset_2$cancer.type.c
```

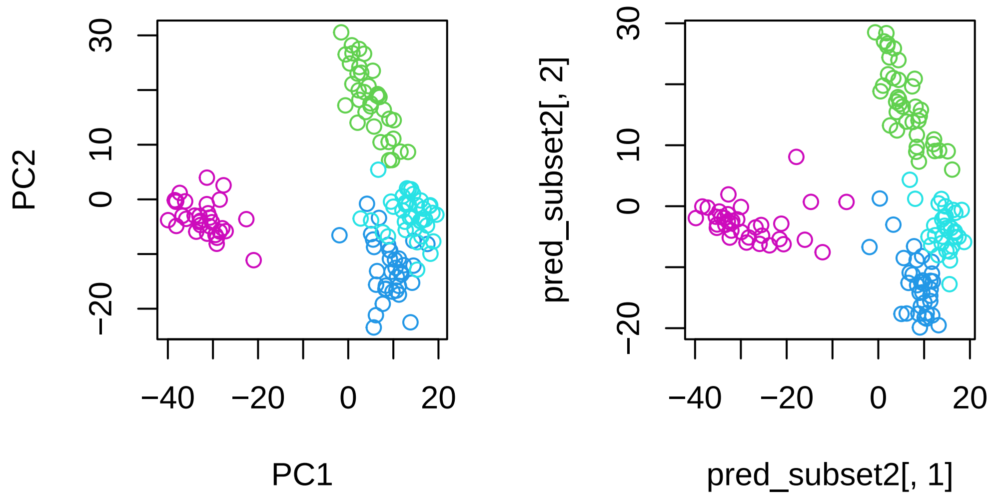


Figure 27.5.: In this plot, we are comparing the subset 1 PCA plot to that produced by projecting the samples from subset 2 into the first two principle components from subset 1.

By projecting the samples from X_2 into the principal component space derived from X_1, we can observe how well the learned principal components capture the structure and variability of the second dataset. This demonstrates the power of transfer learning, where knowledge gained from one dataset can be effectively applied to another related dataset.

Mathematically, transfer learning with PCA can be summarized as follows:

1. Perform PCA on X_1 to obtain the principal components P.
2. Project X_2 into the principal component space using $X_2^{(p)} = X_2 \cdot P$.
3. Compare the distribution of samples from X_1 and X_2 in the principal component space.

Transfer learning with PCA allows us to leverage the learned principal components from one dataset to analyze and understand another related dataset, even when the datasets are collected separately. This technique can be particularly useful when dealing with limited sample sizes or when trying to integrate information from multiple sources.

References

Bourgon, Richard, Robert Gentleman, and Wolfgang Huber. 2010. "Independent Filtering Increases Detection Power for High-Throughput Experiments." *Proceedings of the National Academy of Sciences* 107 (21): 9546–51. https://doi.org/10.1073/pnas.0914005107.

Brouwer-Visser, Jurriaan, Wei-Yi Cheng, Anna Bauer-Mehren, Daniela Maisel, Katharina Lechner, Emilia Andersson, Joel T. Dudley, and Francesca Milletti. 2018. "Regulatory T-Cell Genes Drive Altered Immune Microenvironment in Adult Solid Cancers and Allow for Immune Contextual Patient Subtyping." *Cancer Epidemiology, Biomarkers & Prevention* 27 (1): 103–12. https://doi.org/10.1158/1055-9965.EPI-17-0461.

Buenrostro, Jason D, Paul G Giresi, Lisa C Zaba, Howard Y Chang, and William J Greenleaf. 2013. "Transposition of Native Chromatin for Fast and Sensitive Epigenomic Profiling of Open Chromatin, DNA-binding Proteins and Nucleosome Position." *Nature Methods* 10 (12): 1213–18. https://doi.org/10.1038/nmeth.2688.

Buenrostro, Jason D, Beijing Wu, Howard Y Chang, and William J Greenleaf. 2015. "ATAC-seq: A Method for Assaying Chromatin Accessibility Genome-Wide." *Current Protocols in Molecular Biology / Edited by Frederick M. Ausubel ... [Et Al.]* 109 (January): 21.29.1–9. https://doi.org/10.1002/0471142727.mb2129s109.

Caron, Stéphane. 2018. "The Grammar of Graphics." https://dotlayer.org/en/grammar-of-graphics/.

Center, Pew Research. 2016. "Lifelong Learning and Technology." *Pew Research Center: Internet, Science & Tech*. https://www.pewresearch.org/internet/2016/03/22/lifelong-learning-and-technology/.

Crawford, Gregory E, Sean Davis, Peter C Scacheri, Gabriel Renaud, Mohamad J Halawi, Michael R Erdos, Roland Green, Paul S Meltzer, Tyra G Wolfsberg, and Francis S Collins. 2006. "DNase-chip: A High-Resolution Method to Identify DNase I Hypersensitive Sites Using Tiled Microarrays." *Nature Methods* 3 (7): 503–9. http://www.ncbi.nlm.nih.gov/pubmed/16791207?dopt=AbstractPlus.

Crawford, Gregory E, Ingeborg E Holt, James Whittle, Bryn D Webb, Denise Tai, Sean Davis, Elliott H Margulies, et al. 2006. "Genome-Wide Mapping of DNase Hypersensitive Sites Using Massively Parallel Signature Sequencing (MPSS)." *Genome Research* 16 (1): 123–31. http://www.ncbi.nlm.nih.gov/pubmed/16344561?dopt=AbstractPlus.

DeRisi, J. L., V. R. Iyer, and P. O. Brown. 1997. "Exploring the Metabolic and Genetic Control of Gene Expression on a Genomic Scale." *Science (New York, N.Y.)* 278 (5338):

680–86. https://doi.org/10.1126/science.278.5338.680.

Greener, Joe G., Shaun M. Kandathil, Lewis Moffat, and David T. Jones. 2022. "A Guide to Machine Learning for Biologists." *Nature Reviews Molecular Cell Biology* 23 (1): 40–55. https://doi.org/10.1038/s41580-021-00407-0.

Knowles, Malcolm S., Elwood F. Holton, and Richard A. Swanson. 2005. *The Adult Learner: The Definitive Classic in Adult Education and Human Resource Development*. 6th ed. Amsterdam ; Boston: Elsevier.

Lawrence, Michael, Wolfgang Huber, Hervé Pagès, Patrick Aboyoun, Marc Carlson, Robert Gentleman, Martin T Morgan, and Vincent J Carey. 2013. "Software for Computing and Annotating Genomic Ranges." *PLoS Computational Biology* 9 (8): e1003118. https://doi.org/10.1371/journal.pcbi.1003118.

Libbrecht, Maxwell W., and William Stafford Noble. 2015. "Machine Learning Applications in Genetics and Genomics." *Nature Reviews Genetics* 16 (6): 321–32. https://doi.org/10.1038/nrg3920.

Morgan, Martin, Herve Pages, V Obenchain, and N Hayden. 2016. "Rsamtools: Binary Alignment (BAM), FASTA, Variant Call (BCF), and Tabix File Import." *R Package Version* 1 (0): 677–89.

Student. 1908. "The Probable Error of a Mean." *Biometrika* 6 (1): 1–25. https://doi.org/10.2307/2331554.

Tsompana, Maria, and Michael J Buck. 2014. "Chromatin Accessibility: A Window into the Genome." *Epigenetics & Chromatin* 7 (1): 33. https://doi.org/10.1186/1756-8935-7-33.

Wickham, Hadley. 2014. "Tidy Data." *Journal of Statistical Software, Articles* 59 (10): 1–23. https://doi.org/10.18637/jss.v059.i10.

A. Appendix

A.1. Data Sets

- BRFSS subset
- ALL clinical data
- ALL expression data

A.2. Swirl

The following is from the swirl website.

> The swirl R package makes it fun and easy to learn R programming and data science. If you are new to R, have no fear.

To get started, we need to install a new package into R.

```
install.packages('swirl')
```

Once installed, we want to load it into the R workspace so we can use it.

```
library('swirl')
```

Finally, to get going, start swirl and follow the instructions.

```
swirl()
```

B. Additional resources

- Base R Cheat Sheet

Index