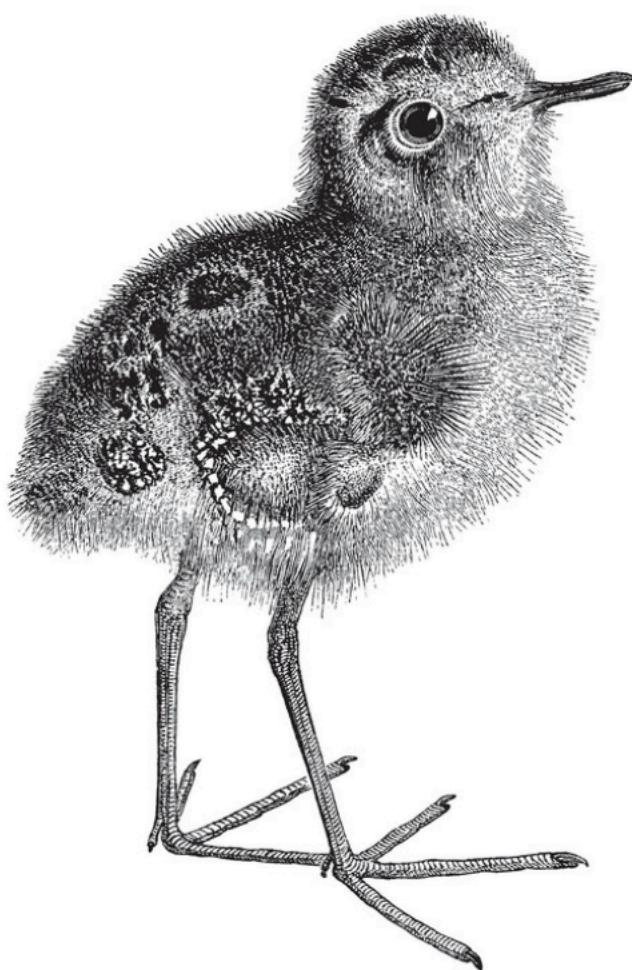


O'REILLY®

Essential Math for Data Science

Take Control of Your Data with Fundamental Linear Algebra, Probability, and Statistics



Early
Release
RAW &
UNEDITED

Thomas Nield

Essential Math for Data Science

Take Control of Your Data with Fundamental Linear
Algebra, Probability, and Statistics

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Thomas Nield

Essential Math for Data Science

by Thomas Nield

Copyright © 2022 Thomas Nield. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North,
Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jessica Haberman

Development Editor: Jill Leonard

Production Editor: Kristen Brown

Copyeditor:

Proofreader:

Indexer:

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator:

October 2022: First Edition

Revision History for the Early Release

- 2021-07-16: First Release

- 2021-09-08: Second Release
- 2021-11-05: Third Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098102937> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Essential Math for Data Science*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-10293-7

[LSI]

Preface

In the past 10 years or so, there has been a growing interest in applying math and statistics to our everyday work and lives. Why is that? Does it have to do with the accelerated interest in “data science” and “big data.” Or is it the promise of machine learning and “artificial intelligence” changing our lives? Media headlines are inundated with stories about studies, polls, and research findings. But these same headlines cause us to ask critical questions, and we are uncertain what these critical questions actually are.

I will make the argument that the disciplines of math and statistics have captured mainstream interest because of the growing availability of data, and we need math and statistics to make sense of it. Yes, we do have scientific tools, machine learning, and other automations that call to us like sirens. We are to blindly trust these “black boxes,” devices and softwares we don’t understand but we use them anyway.

While it is easy to believe computers are smarter than us (and this idea is frequently marketed), the reality cannot be more the opposite. This disconnect can be precarious on so many levels. Do you really want an “algorithm” or “AI” performing criminal sentencing or driving a vehicle, but nobody including the developer can explain why it came to a specific decision? Explainability is the next frontier of statistical computing and AI. This can only begin when we open up the “black box” and uncover the math.

You may also ask how can a developer not know how their own algorithm works? We will talk about that in the second half of the book when we discuss machine learning techniques, and emphasize why we need to understand the math behind the black boxes we build.

To another point, the reason data is being collected on a massive scale is largely due to connected devices and their presence in our everyday lives. We no longer solely use the internet on a desktop or laptop computer. We now take it with us in our smart phones, cars, and household devices. This

has subtly enabled a transition over the past two decades. Data has now evolved from an operational tool to something that is collected and analyzed for less defined objectives. A smartwatch is constantly collecting data on our heart rate, breathing, walking distance, and other markers. Then it uploads that data to a cloud to be analyzed with other users. Our driving habits are being collected by computerized cars, and being used by manufacturers to learn our driving style or enable “self-driving” vehicles. Even “smart toothbrushes” are finding their way into drug stores, which track brushing habits and store that data in a cloud. Whether smart toothbrush data is useful and essential is another discussion!

All of this data collection is permeating every corner of our lives. It can be overwhelming, and a whole book can be written on privacy concerns and ethics. But this availability of data also creates opportunities to leverage math and statistics in new ways, and create more exposure outside academic environments. We can learn more about the human experience, improve product design and application, and optimize commercial strategies. If you understand the ideas presented in this book, you will be able to unlock the value held in our data-hording infrastructure. This does not imply that data and statistical tools are a silver bullet to solve all the world’s problems, but it has given us new tools that we can use.

This growing availability of data has made way for “data science” and “big data” to become demanded professions. We define essential math as an exposure to probability, linear algebra, statistics, and machine learning. If you are seeking a career in data science, machine learning, or engineering, these topics are necessary. I will throw in just enough college math and calculus necessary to better understand what goes in the “black box” libraries you will encounter.

With this book, I aim to give readers an exposure to different mathematical and statistical areas that will be applicable to real-world problems. The first four chapters cover foundational math concepts including practical calculus, probability, linear algebra, and statistics. The last three chapters will segue into machine learning. The ultimate purpose of teaching machine learning is to integrate everything we learn, and demonstrate practical insights in using

machine learning and statistical libraries beyond a “black box” understanding.

The only tool that is needed to follow examples is a Windows/Mac/Linux computer and a Python 3 environment of your choice. The primary Python libraries we will need are `numpy`, `scipy`, `sympy`, and `sklearn`. If you are unfamiliar with Python, it is a friendly and easy-to-use programming language with massive learning resources behind it. Here are some I recommend:

- *Data Science from Scratch 2nd Edition (O'Reilly)* by Joel Grus - The second chapter of this book has the best crash course in Python I have encountered. Even if you have never written code before, Joel does a fantastic job getting you up and running with Python effectively in the shortest time possible. It is also a great book to have on your shelf and to apply your mathematical knowledge!
- *Python for the Busy Java Developer (Apress)* by Deepak Sarda - If you are a software engineer coming from a statically-typed, object-oriented programming background, this is the book to grab. As someone who started programming with Java, I have a deep appreciation how Deepak shares Python features and relates them to Java developers. If you have done .NET, C++, or other C-like languages you will probably learn Python effectively from this book as well.

This book will not make you an expert or give you PhD knowledge. I avoid mathematical expressions full of Greek symbols, and instead strive to use plain English in its place. But, what this book will do is make you more comfortable talking about math and statistics, giving you *essential* knowledge to navigate these areas successfully. I believe the widest path to success is not having deep, specialized knowledge in one topic, but instead having exposure and practical knowledge across several topics. That is the goal of this book, and you will learn just enough to be dangerous and ask those once elusive critical questions.

So let's get started!

Chapter 1. Basic Math and Calculus Review

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at thomasnield@live.com.

We will kick off the first chapter covering what numbers are and how variables and functions work on a Cartesian system. We will then cover exponents and logarithms. After that we will learn the two basic operations of calculus: derivatives and integrals.

Before we dive into the applied areas of essential math such as probability, linear algebra, statistics, and machine learning, we should probably review a few basic math and calculus concepts. Before you drop this book and run screaming, do not worry! I will present how to calculate slopes and areas for a function in a way you were probably not taught in college. We got Python on our side, not a pencil and paper.

I will make these topics as tight and practical as possible, focusing only on what will help us in later chapters and fall under the “essential math” umbrella.

THIS IS NOT A FULL MATH CRASH COURSE!

This is by no means a comprehensive review of high school and college math. If you want that, a great book to check out is *No Bullshit Guide to Math and Physics* by Ivan Savov. The first few chapters contain the best crash course on high school and college math I have ever seen. The book *Mathematics 1001* by Dr. Richard Elwes has some great content as well, and in bite-sized explanations.

Number Theory

What are numbers? I promise to not be too philosophical in this book, but are numbers not a construct we have defined? Why do we have the digits 0 through 9, and not have more digits than that? Why do we have fractions and decimals and not just whole numbers? This area of math where we muse about numbers and why we designed them a certain way is known as number theory.

Number theory goes all the way back to ancient times, where mathematicians study different number systems and why we have accepted them the way we do today. Here are different number systems that you may recognize:

Natural Numbers

These are the numbers 1, 2, 3, 4, 5... and so on. Only positive numbers are included here, and are the earliest known system. Natural numbers are so ancient cavemen scratched tally marks on bones and cave walls to keep records.

Whole Numbers

Adding to natural numbers, the concept of “0” was later accepted and we call these “whole numbers.” The Babylonians also developed the useful idea for place-holding notation for empty “columns” on numbers greater than 9, such as “10”, “1000”, or “1090.” Those zeros indicate no value occupying that column.

Integers

Integers include positive and negative whole numbers as well as 0. We may take them for granted, but ancient mathematicians were deeply distrusting of the idea of negative numbers. But when you subtract 5 from 3, you get -2. This is useful especially when it comes to finances where we measure profits and losses. In 628 AD, an Indian mathematician named Brahmagupta showed why negative numbers were necessary for arithmetic to progress, and therefore integers became accepted.

Rational Numbers

Any number that you can express as a fraction, such as $\frac{2}{3}$, is a rational number. This includes all finite decimals and integers since they can be expressed as fractions too, such as $\frac{687}{100} = 6.87$ and $\frac{2}{1} = 2$ respectively. They are called *rational* because they are *ratios*. Rational numbers were quickly deemed necessary because time, resources, and other quantities could not always be measured in discrete units. Milk does not always come in gallons. We may have to measure it as parts of a gallon. If I run for 12 minutes, I cannot be forced to measure in whole miles when in actuality I ran $\frac{9}{10}$ of a mile.

Irrational Numbers

Irrational numbers cannot be expressed as a fraction. This includes the famous Pi π , square roots of certain numbers like $\sqrt{2}$, and Euler's number e which we will learn about later. These numbers have an infinite number of decimal digits, such as $\pi = 3.141592653589793238462\dots$

There is an interesting history behind irrational numbers. The Greek mathematician Pythagoras believed all numbers are rational. He believed this so fervently, he made a religion that prayed to the number 10. "*Bless us, divine number, thou who generated gods and men!*" he and his followers would pray (why "10" was so special, I do not know). There is a legend that one of his followers Hippasus proved not all numbers are rational simply by demonstrating the square root of 2. This severely

messed with Pythagoras' belief system, and he responded by drowning Hippasus out at sea.

Regardless, we now know not all numbers are rational.

Real Numbers

Real numbers include rational as well as irrational numbers. In practicality, when you are doing any data science work you can treat any decimals you work with as real numbers.

Complex and Imaginary Numbers

You encounter this number type when you take the square root of a negative number. While imaginary and complex numbers have relevance in certain types of problems, we will mostly steer clear from them for our purposes

In data science, you will find most (if not all) your work will be using natural numbers, integers, and real numbers. Imaginary numbers may be encountered in more advanced use cases such as matrix decomposition, which we will touch on in Chapter 4.

COMPLEX AND IMAGINARY NUMBERS

If you do want to learn about imaginary numbers, there is a great playlist *Imaginary Numbers are Real* on YouTube here: <https://youtu.be/T647CGsuOVU>.

Order of Operations

Hopefully you are familiar with **order of operations** which is the order you solve each part of a mathematical expression. As a brief refresher, recall you evaluate components in parentheses, followed by exponents, then multiplication, division, addition, and subtraction. You can remember the order of operations by the mnemonic device **PEMDAS** (Please Excuse My

Dear Aunt Sally) which corresponds to the ordering parenthesis, exponents, multiplication, division, addition, and subtraction.

Take for example this expression:

$$2 * \frac{(3 + 2)^2}{5} - 4$$

First we evaluate the parentheses $(3 + 2)$ which evaluates to 5.

$$2 * \frac{5^2}{5} - 4$$

Next we solve the exponent, which we can see is squaring that 5 we just summed. That is 25.

$$2 * \frac{25}{5} - 4$$

Next up we have multiplication and division. The ordering of these two is swappable since division is also multiplication (using fractions). Let's go ahead and multiply the 2 with the $\frac{25}{5}$, yielding $\frac{50}{5}$

$$\frac{50}{5} - 4$$

Next we will perform the division, dividing 50 by 5 which will yield 10.

$$\mathbf{10 - 4}$$

And finally we perform any addition and subtraction. Of course, $10 - 4$ is going to give us 6.

$$10 - 4 = 6$$

Sure enough, if we were to express this in Python we would print a value of 6.0 as shown in [Example 1-1](#).

Example 1-1. Solving an expression in Python

```
my_value = 2 * (3 + 2)**2 / 5 - 4  
  
print(my_value) # prints 6.0
```

This may be elementary to you, but it is still critical nonetheless. In code, even if you get the correct result without them, it is a good practice to liberally use parenthesis in complex expressions so you establish control of the evaluation order.

Here I group up the fractional part of my expression in parenthesis, helping to set it apart from the rest of the expression in [Example 1-2](#).

Example 1-2. Making use of parenthesis for clarity in Python

```
my_value = 2 * ((3 + 2)**2 / 5) - 4  
  
print(my_value) # prints 6.0
```

While both examples are technically correct, the latter one is more clear to us easily confused humans. If you or someone else makes changes to your code, the parenthesis provide an easy reference of operation order as you make changes. This provides a line of defense against code changes to prevent bugs as well.

Variables

If you have done some scripting with Python or another programming language, you have an idea what a variable is. In mathematics, a **variable** is a named placeholder for an unspecified or unknown number.

You may have a variable x representing any real number, and you can multiply that variable without declaring what it is. In [Example 1-3](#) we take a variable input x from a user, and multiply it by 3.

Example 1-3. A variable in Python that is then multiplied

```
x = int(input("Please input a number\n"))  
  
product = 3 * x  
  
print(product)
```

There are some standard variable names for certain variable types. If these variable names and concepts are unfamiliar, no worries! But the rest of you readers might recognize we use theta θ to denote angles and beta β for a parameter in a linear regression. Greek symbols make awkward variable names in Python, so we would likely name these variables `theta` and `beta` in Python as shown in [Example 1-4](#).

Example 1-4. Greek variable names in Python

```
beta = 1.75  
theta = 30.0
```

Note also that variable names can be subscripted so that several instances of a variable name can be used. For practical purposes, just treat these as separate variables. If you encounter variables x_1 , x_2 , and x_3 , just treat them as three separate variables as shown in [Example 1-5](#).

Example 1-5. Expressing subscripted variables in Python

```
x1 = 3 # or x_1 = 3  
x2 = 10 # or x_2 = 10  
x3 = 44 # or x_3 = 44
```

Functions

Functions are expressions that define relationships between two or more variables. More specifically, a function takes **input variables** (also called **domain variables** or **independent variables**), plugs them into an expression, and then results in an **output variable** (also called **dependent variable**).

Take this simple linear function $y = 2x + 1$. For any given x value, we solve the expression with that x to find y . When $x = 1$, then $y = 3$. When $x = 2$, $y = 5$. When $x = 3$, $y = 7$ and so on as shown in [Table 1-1](#).

$$y = 2x + 1$$

Table 1-1. Different values for $y = 2x + 1$

x	$2x + 1$	y
0	$2(0) + 1$	1
1	$2(1) + 1$	3
2	$2(2) + 1$	5
3	$2(3) + 1$	7

Functions are useful because they help predict the relationship between variables, such as how many fires y can we expect at x temperature. We will use linear functions to perform linear regressions in [Chapter 5](#).

Another convention you may see for the dependent variable y is to explicitly label it a function of x , such as $f(x)$. So rather than express a function as $y = 2x + 1$ we can also express it as:

$$f(x) = 2x + 1$$

[Example 1-6](#) shows how we can declare a mathematical function and iterate it in Python.

Example 1-6. Declaring a linear function in Python

```
def f(x):
    return 2 * x + 1

x_values = [0, 1, 2, 3]

for x in x_values:
    y = f(x)
    print(y)
```

When dealing with real numbers, a subtle but important feature of functions is they often have an infinite number of x values and resulting y values. Ask yourself this: how many x values can we put through the function $y = 2x + 1$? Rather than just 0, 1, 2, 3... why not 0, .5, 1, 1.5, 2, 2.5, 3 as shown in [Table 1-2](#) below?

Table 1-2. Different values for $y = 2x + 1$

x	$2x + 1$	y
0.0	$2(0) + 1$	1
0.5	$2(.5) + 1$	2
1.0	$2(1) + 1$	3
1.5	$2(1.5) + 1$	4
2.0	$2(2) + 1$	5
2.5	$2(2.5) + 1$	6
3.0	$2(3) + 1$	7

Or why not do quarter steps for x ? $\frac{1}{10}$ of a step? We can make these steps infinitely small effectively showing $y = 2x + 1$ is a **continuous function**, where for every possible value of x there is a value for y . This segues us nicely to visualize our function as a line as shown in [Figure 1-1](#).

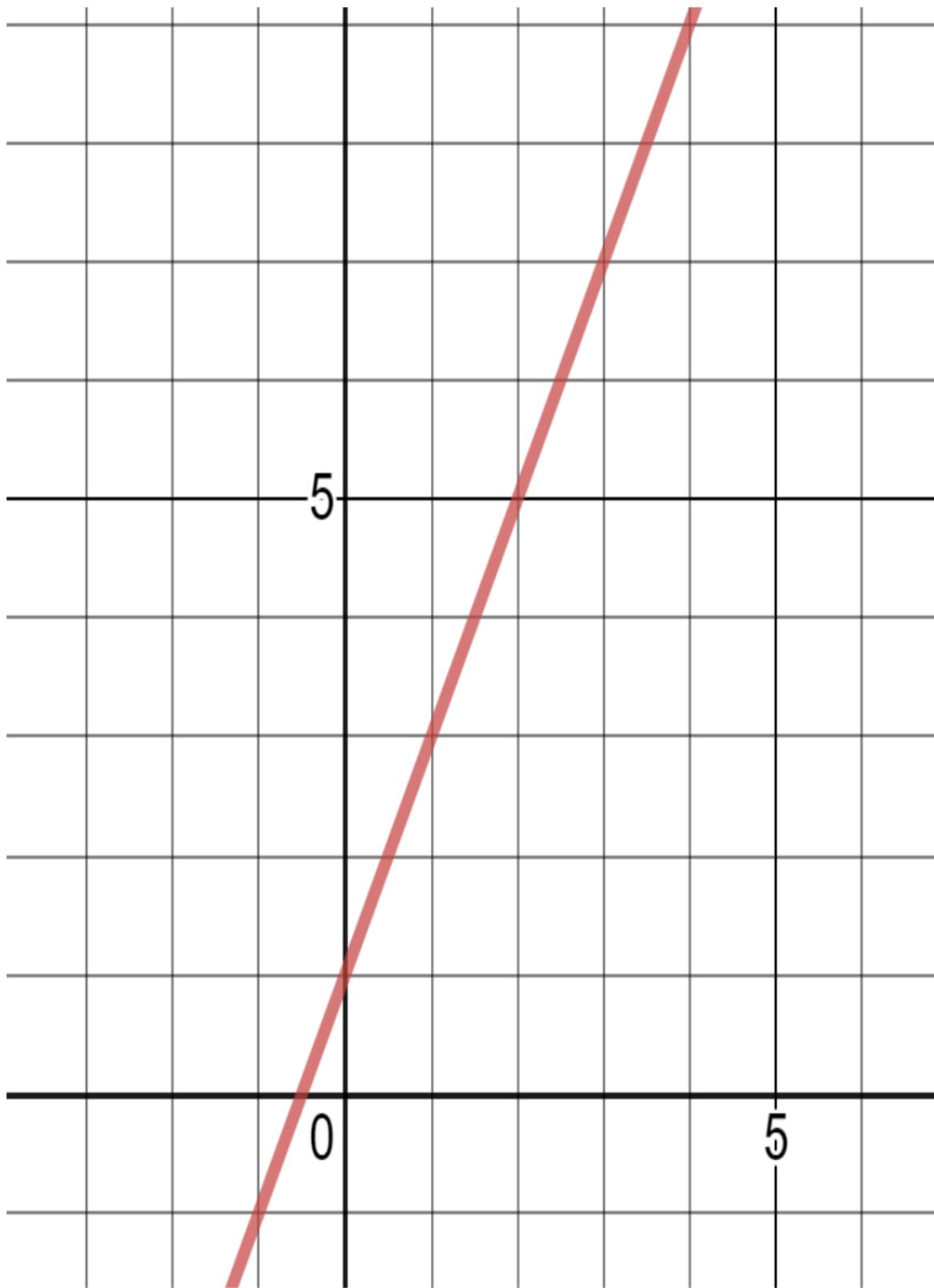


Figure 1-1. Graph for function $y = 2x + 1$

When we plot on a two-dimensional plane with two number lines (one for each variable) it is known as a **Cartesian plane**, **x-y plane**, or **coordinate plane**. We trace a given x value and then look up the corresponding y value, and plot the intersections as a line. Notice that due to the nature of real numbers (or decimals, if you prefer), there are an infinite number of x values. This is why when we plot the function $f(x)$ we get a continuous line with no breaks in it. There are an infinite number of points on that line, or any part of that line.

If you want to plot this using Python, there are a number of charting libraries from Plotly to matplotlib. However, SymPy gives us a quick, clean way to plot a function. It uses matplotlib so make sure you have that package installed, otherwise it will print an ugly text-based graph to your console. After that, just declare the x variable to SymPy using `symbols()`, declare your function, and then plot it as shown in [Example 1-7](#) and [Figure 1-2](#).

Example 1-7. Charting a linear function in Python using SymPy

```
from sympy import *
x = symbols('x')
f = 2*x + 1
plot(f)
```

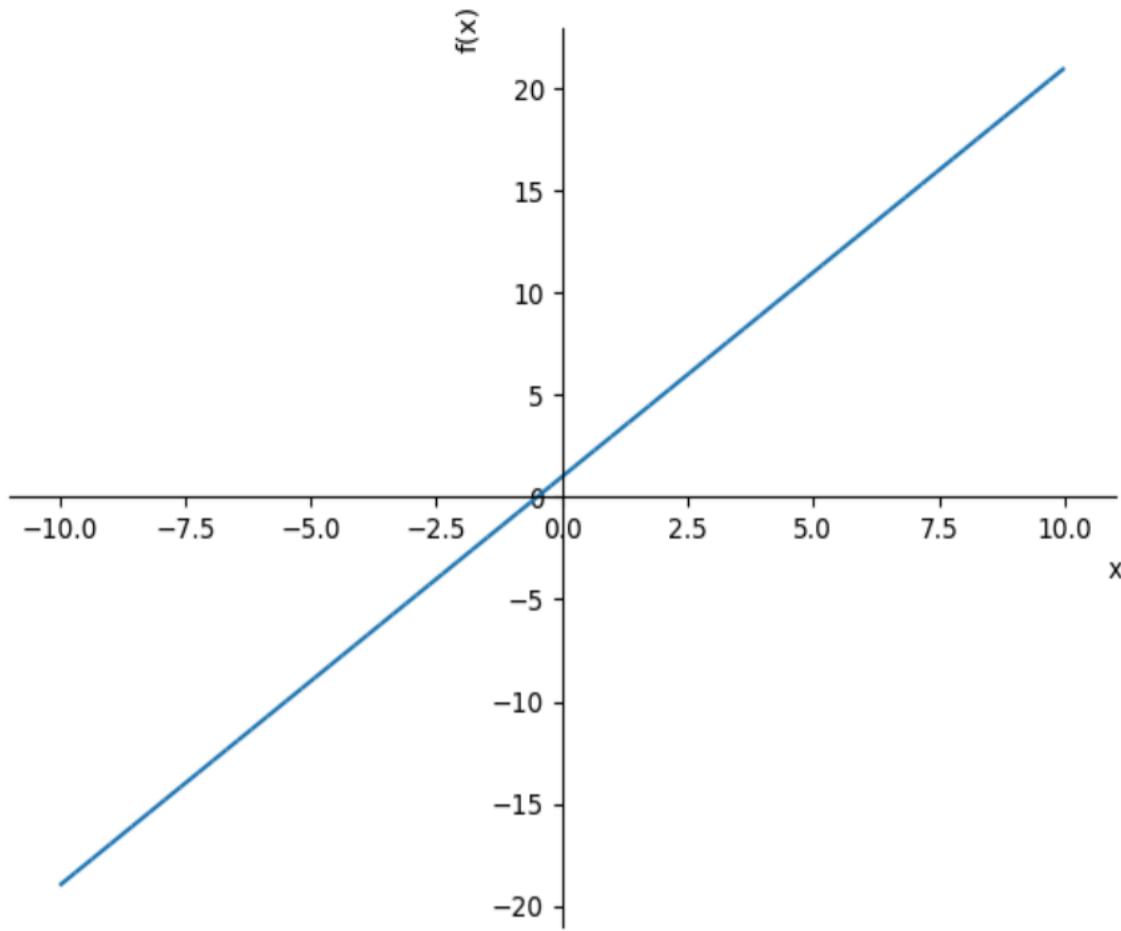


Figure 1-2. Using SymPy to graph a linear function

Example 1-8 and Figure 1-3 are another example showing the function $y = x^2 + 1$.

Example 1-8. Charting a linear function in Python using SymPy

```
from sympy import *
```

```
x = symbols('x')
f = x**2 + 1
plot(f)
```

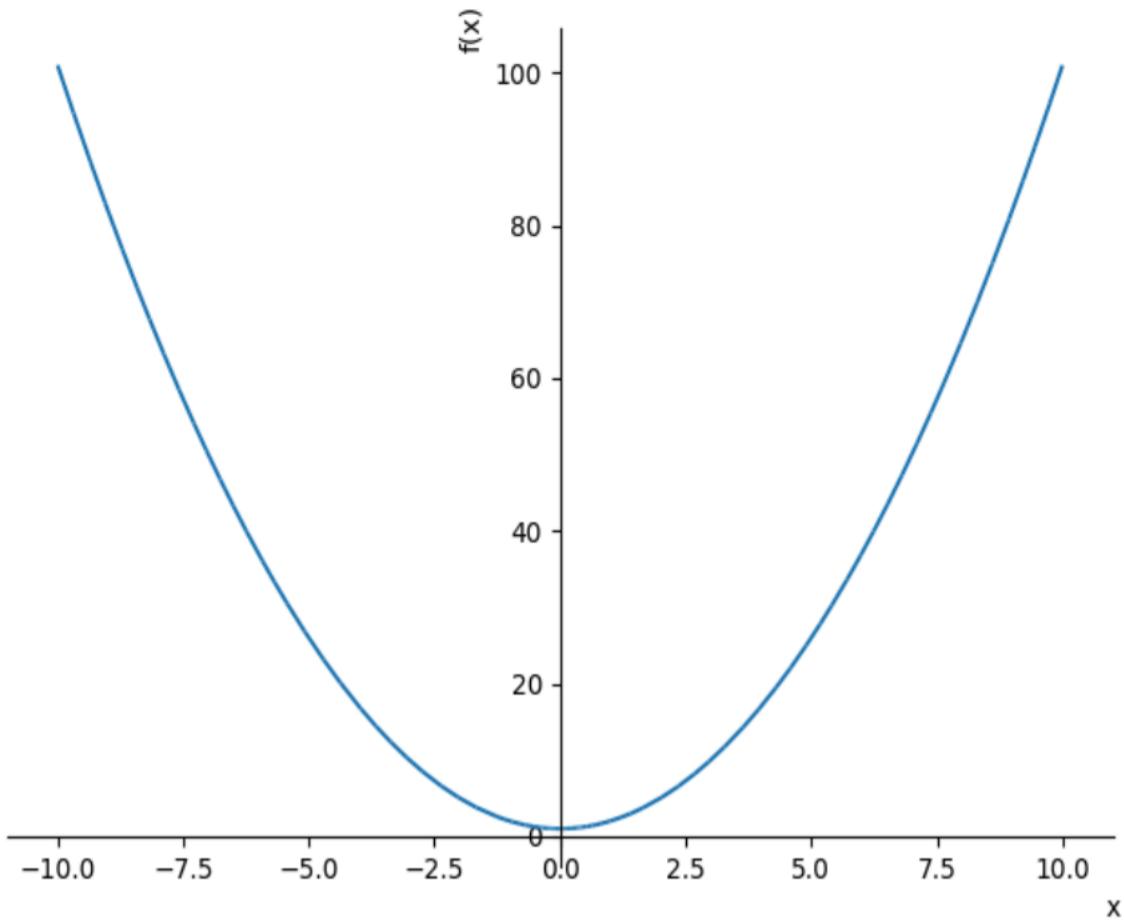


Figure 1-3. Using SymPy to graph an exponential function

Notice in Figure 1-3 we do not get a straight line but rather a smooth, symmetrical curve known as a parabola. It is continuous but not linear, as it does not produce values in a straight line. Curvy functions like this are mathematically harder to work with, but we will learn some tricks to make it not so bad.

CURVILINEAR FUNCTIONS

When a function is continuous but curvy, rather than linear and straight, we call it a *curvilinear function*.

Note that functions can utilize multiple input variables and not just one. For example, we can have a function with independent variables x and y . Note

that y is not dependent like in previous examples.

$$f(x, y) = 2x + 3y$$

Since we have 2 independent variables (x and y) and 1 dependent variable (the output of $f(x, y)$) we need to plot this graph on 3 dimensions to produce a plane of values rather than a line as shown in [Example 1-9](#) and [Figure 1-4](#).

Example 1-9. Declaring a function with 2 independent variables in Python:

```
from sympy import *
from sympy.plotting import plot3d

x, y = symbols('x y')
f = 2*x + 3*y
plot3d(f)
```

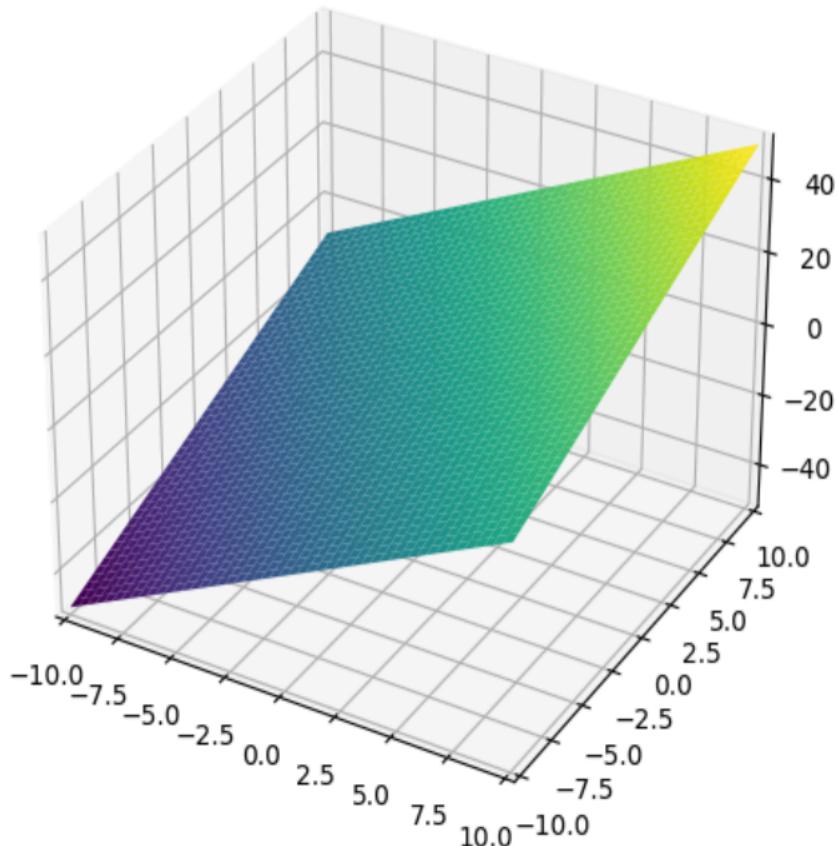


Figure 1-4. Using SymPy to graph a 3-dimensional function

No matter how many independent variables you have, your function will typically only output one dependent variable. When you solve for multiple dependent variables, you will likely be using separate functions for each one.

Summations

I promised not to use equations full of Greek symbols in this book. However, there is one that is so common and useful that it would be remiss to not cover it. A **summation** is expressed as a sigma Σ and adds elements together.

For example, if I wanted to iterate the numbers 1 through 5, multiply each by 2, and sum them here is how I would express that using a summation.

Example 1-10 shows how to execute this in Python.

$$\sum_{i=1}^5 2i = (2)1 + (2)2 + (2)3 + (2)4 + (2)5 = 30$$

Example 1-10. Performing a summation in Python

```
summation = sum(2*i for i in range(1,6))
print(summation)
```

Note that i is a placeholder variable representing each consecutive index value we are iterating in the loop, which we multiply by 2 and then sum all together. When you are iterating data, you may see variables like x_i indicating an element in a collection at index i .

THE RANGE() FUNCTION

Recall that the `range()` function in Python is end exclusive, meaning if you invoke `range(1,4)` it will iterate the numbers 1, 2, and 3. It excludes the 4 as an upper boundary.

It is also common to see n represent the number of items in a collection, like the number of records in a dataset. Here is one such example where we

iterate a collection of numbers of size n , multiply each one by 10, and sum them.

$$\sum_{i=1}^n 10x_i$$

In [Example 1-11](#) we use Python we execute this expression on a collection of 4 numbers. Note that in Python (and most programming languages in general) we typically reference items starting at index 0, while in math we start at index 1. Therefore we shift accordingly in our iteration by starting at 0 in our `range()`.

Example 1-11. Summation of elements in Python

```
x = [1, 4, 6, 2]
n = len(x)

summation = sum(10*x[i] for i in range(0,n))
print(summation)
```

That is the gist of summation. In a nutshell, a summation Σ says “add a bunch of things together” and uses an index i and a maximum value n to express each iteration feeding into the sum. We will see them in almost every other chapter in this book.

SUMMATIONS IN SYMPY

Later in this chapter we will learn about SymPy, a symbolic math library. Here's a sidebar for reference. A summation operation in SymPy is performed using the `Sum()` operator. Below in [Example 1-12](#) we iterate i from 1 through n , multiply each i , and sum them. But then we use the `subs()` function to specify n as 5, which will then iterate and sum all i elements from 1 through n .

Example 1-12.

```
from sympy import *

i,n = symbols('i n')

# iterate each element i from 1 to n,
# then multiply and sum
summation = Sum(2*i,(i,1,n))

# specify n as 5,
# iterating the numbers 1 through 5
up_to_5 = summation.subs(n, 5)
print(up_to_5.doit()) # 30
```

Note that summations in SymPy are lazy, meaning they do not automatically calculate or get simplified. So use the `doit()` function to execute the expression.

Exponents

Exponents multiplies a number by itself a specified number of times. When you raise 2 to the 3rd power (expressed as 2^3 using 3 as a superscript), that is multiplying three 2's together:

$$2^3 = 2 \cdot 2 \cdot 2 = 8$$

The **base** is the variable or value we are exponentiating, and the **exponent** is the number of times we multiply the base value. For the expression 2^3 , 2 is

the base and 3 is the exponent.

Exponents have a few interesting properties. Say we multiplied x^2 and x^3 together. Observe what happens below when I expand the exponents with simple multiplication, and then consolidate into a single exponent:

$$x^2 x^3 = (x*x)*(x*x*x) = x^{2+3} = x^5$$

When we multiply exponents together with the same base, we simply add the exponents which is known as the **product rule**. Note that the base of all multiplied exponents must be the same for the product rule to apply.

Let's explore division next. What happens when we divide x^2 by x^5 ?

$$\begin{array}{r} x^2 \\ \hline x^5 \\ \hline x^*x \\ \hline x^*x^*x^*x^*x \\ \hline 1 \\ \hline x^*x^*x \\ \hline \frac{1}{x^3} = x^{-3} \end{array}$$

As you can see above, when we divide x^2 by x^5 we can cancel out two x 's in numerator and denominator, leaving us with $\frac{1}{x^3}$. Without tangentialing into algebra rules too much, here is what happens. When a factor exists in both the numerator and denominator, we can cancel out that factor.

This is a good point to introduce negative exponents, which is another way of expressing an exponent operation in the denominator of a fraction. To demonstrate, $\frac{1}{x^3}$ is the same as x^{-3} .

$$\frac{1}{x^3} = x^{-3}$$

Tying back the product rule, we can see it applies to negative exponents too. To get intuition behind this, let's approach this problem a different way. We can express this division of two exponents by making the “5” exponent of x^5 negative, and then multiplying it with x^2 . When you add a negative number, it is effectively performing subtraction. Therefore, the exponent product rule summing the multiplied exponents still holds up as shown below:

$$\frac{x^2}{x^5} x^2 \frac{1}{x^5} = x^2 x^{-5} = x^{2+(-5)} = x^{-3}$$

SIMPLIFY EXPRESSIONS WITH SYMPY

If you get uncomfortable with simplifying algebraic expressions, you can use the SymPy library to do the work for you. [Example 1-13](#) shows how to simplify our previous example.

Example 1-13.

```
from sympy import *
x = symbols('x')
expr = x**2 / x**5
print(expr) # x**(-3)
```

Now what about fractional exponents? They are just an alternative way to represent roots, such as the square root. As a brief refresher, a square root of 4 asks “what number multiplied by itself will give me 4?”, which of course is 2. Note here that $4^{1/2}$ is the same as $\sqrt{4}$.

$$4^{1/2} = \sqrt{4} = 2$$

Cubed roots are similar to square roots, but they seek a number multiplied by itself 3 times to give a result. A cubed root of 8 is expressed as $\sqrt[3]{8}$, and asks “what number multiplied by itself 3 times gives me 8”? This number would be 2 because $2 \times 2 \times 2 = 8$. In exponents, a cubed root is expressed as a fractional exponent, and $\sqrt[3]{8}$ can be re-expressed as $8^{1/3}$.

$$8^{1/3} = \sqrt[3]{8} = 2$$

To bring it back full circle, what happens when you multiply the cubed root of 8 three times? This will undo the cubed root and yield us 8. Alternatively, if we express the cubed root as fractional exponents $8^{1/3}$, it becomes clear we add the exponents together to get an exponent of 1. That also undoes the cubed root.

$$\sqrt[3]{8} * \sqrt[3]{8} * \sqrt[3]{8} = 8^{\frac{1}{3}} * 8^{\frac{1}{3}} * 8^{\frac{1}{3}} = 8^{\frac{1}{3} + \frac{1}{3} + \frac{1}{3}} = 8^1 = 8$$

And one last property: with an exponent of an exponent, that would multiply the exponents together. This is known as the **power rule**. So $(8^3)^2$ would simplify to 8^6 .

$$(8^3)^2 = 8^{3*2} = 8^6$$

If you are skeptical why this is, try expanding it and you will see the sum rule makes it clear:

$$(8^3)^2 = 8^3 8^3 = 8^{3+3} = 8^6$$

Lastly, what does it mean when we have a fractional exponent with a numerator other than 1, such as $8^{\frac{2}{3}}$? Well that is taking the cube root of 8 and then squaring it. Take a look below:

$$8^{\frac{2}{3}} = \left(8^{\frac{1}{3}}\right)^2 = 2^2 = 4$$

And yes, irrational numbers can serve as exponents like 8^π which is 687.2913. This may feel unintuitive, and understandably so! In the interest of time we will not dive deep into this as it requires some Calculus. But essentially, we can calculate irrational exponents by approximating with a rational number. This is effectively what computers do since they can only compute to so many decimal places anyway.

For example Pi π has an infinite number of decimal places. But if we take the first 11 digits, 3.1415926535, we can approximate Pi as a rational number $31415926535 / 10000000000$. Sure enough, this gives us approximately 687.2913 which should approximately match any calculator.

$$8^{\pi} \approx 8^{\frac{31415926535}{10000000000}} \approx 687.2913$$

Logarithms

A **logarithm** is a math function that finds a power for a specific number and base. It may not sound interesting at first, but it actually has many applications. From measuring earthquakes to managing volume on your stereo, the logarithm is found everywhere. It also finds its way into machine learning and data science a lot. As a matter of fact, logarithms will be a key part of logistic regressions in Chapter 6.

Start your thinking by asking “2 raised to *what power* gives me 8?” One way to express this mathematically is to use an x for the exponent:

$$2^x = 8$$

We intuitively know the answer, $x = 3$, but we need a more elegant way to express this common math operation. This is what the $\log()$ function is for.

$$\log_2 8 = x$$

As you can see in the logarithm expression above, we have a base 2 and are finding a power to give us 8. More generally, we can re-express a variable exponent as a logarithm:

$$a^x = b$$

$$\log_a b = x$$

Algebraically speaking, this is a way of isolating the x which is important to solve for x .

Example 1-14 shows how we calculate this logarithm in Python.

Example 1-14. Using the log function in Python:

```
from math import log

# 2 raised to what power gives me 8?
x = log(8, 2)

print(x) # prints 3.0
```

When you do not supply a base argument to a `log()` function on a platform like Python, it will typically have a default base. In some fields, like earthquake measurements, the default base for the log is 10, but in data science, the default base for the log is Euler's number e . Python uses the latter and we will talk about e shortly.

Just like exponents, logarithms have several properties when it comes to multiplication, division, exponentiation, and so on. In the interest of time and focus, I will just present this table below in [Table 1-3](#). The key idea to focus on is a logarithm finds an exponent for a given base to result in a certain number.

If you need need to dive into logarithmic properties, [Table 1-3](#) displays exponent and logarithm behaviors side-by-side you can use for reference.

Table 1-3. Properties for Exponents and Logarithms

Operator	Exponent Property	Logarithm Property
Multiplication	$x^m * x^n = x^{m+n}$	$\log(a * b) = \log(a) + \log(b)$
Division	$\frac{x^m}{x^n} = x^{m-n}$	$\log\left(\frac{a}{b}\right) = \log(a) - \log(b)$
Exponentiation	$(x^m)^n = x^{mn}$	$\log(a^n) = n * \log(a)$
Zero Exponent	$x^0 = 1$	$\log(1) = 0$
Inverse	$x^{-1} = \frac{1}{x}$	$\log(x^{-1}) = \log(\frac{1}{x}) = -\log(x)$

Euler's Number and Natural Logarithms

There is a special number that shows up quite a bit in math called Euler's number e . It is a special number much like Pi π , and is approximately 2.71828. e is used a lot because it mathematically simplifies a lot of problems.

Back in high school, my Calculus teacher demonstrated Euler's number in several exponential problems. Finally I asked "Mr. Nowe, what is e anyway? Where does it come from?" I remember never being fully satisfied with the explanations involving rabbit populations and other natural phenomena. I hope to give a more satisfying explanation here.

WHY EULER'S NUMBER IS USED SO MUCH

Another property of Euler's number is its exponential function is a derivative to itself, which is convenient for exponential and logarithmic functions. We will learn about derivatives later in this chapter. In many applications where the base does not really matter, we pick the one that results in the simplest derivative, and that is Euler's number. That is also why it is the default base in many data science functions.

Here is how I like to discover Euler's number. Let's say you loan \$100 to somebody with 20% interest annually. Typically interest will be compounded monthly, so the interest each month would be $.20/12 = .01666$. How much will the loan balance be after two years? To keep it simple, let's assume the loan does not require payments (and no payments are made) until the end of those two years.

Putting together the exponent concepts we learned so far (or perhaps pulling it out a finance textbook), we can come up with a formula to calculate interest. It consists of a balance A for a starting investment P , interest rate r , time span t (number of years), and periods n (number of months in each year). Here it is as follows:

$$A = P \times \left(1 + \frac{r}{n}\right)^{nt}$$

So if we were to compound interest every month, the loan would grow to \$148.69 as calculated below:

$$A = P^* \left(1 + \frac{r}{n}\right)^{nt}$$
$$100^* \left(1 + \frac{.20}{12}\right)^{12*2} = 148.6914618$$

If you want to do this in Python, try it out with the code in [Example 1-15](#).

Example 1-15. Calculating compound interest in Python

```
from math import exp

p = 100
r = .20
t = 2.0
n = 12

a = p * (1 + (r/n))**(n * t)

print(a) # prints 148.69146179463576
```

But what if we compounded interest daily? What happens then? Change n to 365.

$$A = P^* \left(1 + \frac{r}{n}\right)^{nt}$$
$$100^* \left(1 + \frac{.20}{365}\right)^{365*2} = 149.1661279$$

Huh! If we compound our interest daily instead of monthly, we would earn 47.4666 cents more at the end of two years. If we got greedy why not compound every hour as shown below? Will that give us even more? There are 8760 hours in a year, so set n to that value:

$$A = P^* \left(1 + \frac{r}{n}\right)^{nt}$$

$$100 * \left(1 + \frac{.20}{8760}\right)^{8760*2} = 149.1817886$$

Ah, we squeezed out roughly 2 cents more in interest! But are we experiencing a diminishing return? Let's try to compound every minute! Note that there are 525,600 minutes in a year, so let's set that value to n :

$$A = P * \left(1 + \frac{r}{n}\right)^{nt}$$

$$100 * \left(1 + \frac{.20}{525600}\right)^{525600*2} = 149.1824584$$

Okay we are only gaining smaller and smaller fractions of a cent the more frequently we compound. So if I keep making these periods infinitely smaller to the point of compounding continuously, where does this lead to?

Let me introduce you to Euler's number e , which is approximately 2.71828. Here is the formula to compound "continuously," meaning we are compounding nonstop:

$$A = P * e^{rt}$$

Returning to our example, let's calculate the balance of our loan after 2 years if we compounded continuously:

$$A = P * e^{rt}$$

$$A = 100 * e^{.20*2} = 149.1824698$$

Okay this is not too surprising considering compounding every minute got us a balance of 149.1824584. That got us really close to our value of 149.1824698 when compounding continuously.

Typically you use e as an exponent base in Python, Excel, and other platforms using the `exp()` function. You will find that e is so commonly used, it is the default base for both exponent and logarithm functions.

Example 1-16 calculates continuous interest in Python using the `exp()` function.

Example 1-16. Calculating continuous interest in Python

```
from math import exp

p = 100 # principal, starting amount
r = .20 # interest rate, by year
t = 2.0 # time, number of years

a = p * exp(r*t)

print(a) # prints 149.18246976412703
```

So where do we derive this constant e ? Compare the compounding interest formula and the continuous interest formula. They structurally look similar, but with some differences:

$$A = P^* \left(1 + \frac{r}{n}\right)^{nt}$$

$$A = P^* e^{rt}$$

More technically speaking, e is the resulting value of the expression $\left(1 + \frac{1}{n}\right)^n$ as n forever gets bigger and bigger, thus approaching infinity. Try experimenting with increasingly large values for n . By making it larger and larger you will notice something:

$$\left(1 + \frac{1}{n}\right)^n$$

$$\left(1 + \frac{1}{100}\right)^{100} = 2.70481382942$$

$$\left(1 + \frac{1}{1000}\right)^{1000} = 2.71692393224$$

$$\left(1 + \frac{1}{10000}\right)^{10000} = 2.71814592682$$

$$\left(1 + \frac{1}{10000000}\right)^{10000000} = 2.71828169413$$

As you make n larger, there is a diminishing return and it converges approximately on the value 2.71828, which is our value e . You will find this number e used not just in studying populations and their growth. It plays a key role in many areas of mathematics.

Later in the book, we will use it to build normal distributions in [Chapter 3](#) and logistic regressions in Chapter 6.

Natural Logarithms

When we use e as our base for a logarithm, we call it a **natural logarithm**. Depending on the platform, we may use `ln()` instead of `log()` to specify a natural logarithm. So rather than express a natural logarithm expressed as $\log_e 10$ to find the power raised on e to get 10, we would shorthand it as `ln(10)`.

$$\log_e 10 = \ln(10)$$

However in Python, a natural logarithm is specified by the `log()` function. As said earlier, the default base for the `log()` function is e . Just leave the second argument for the base empty and it will default to using e as the base shown in [Example 1-17](#).

Example 1-17. Calculating natural logarithm of 10 in Python

```
from math import log

# e raised to what power gives us 10?
x = log(10)

print(x) # prints 2.302585092994046
```

We will use e in a number of places throughout this book. Feel free to experiment with exponents and logarithms for awhile using Excel, Python, Desmos.com, or any other calculation platform of your choice. Make graphs and get comfortable with functions!

Limits

As we have seen with Euler's number, some interesting ideas emerge when we forever increase or decrease an input variable and the output variable keeps approaching a value but never reaching it. Let's formally explore this idea.

Take this function which is plotted in [Figure 1-5](#):

$$f(x) = \frac{1}{x}$$

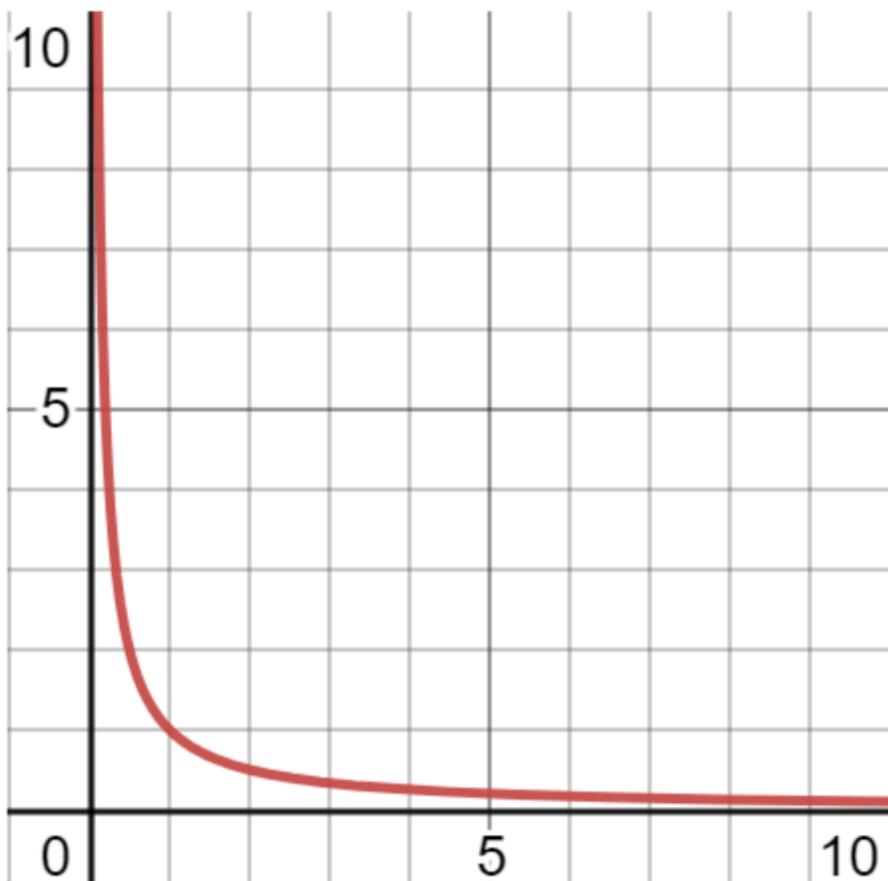


Figure 1-5. A function that forever approaches 0 in either direction but never reaches 0

We are only looking at positive x values. Notice that as x forever increases $f(x)$ gets closer to 0. Fascinatingly, $f(x)$ never actually reaches 0. It just forever keeps getting closer.

Therefore, the fate of this function is as x forever extends into infinity, it will keep getting closer to 0 but never reach 0. The way we express a value that is forever being approached, but never reached, is through a limit.

$$\lim_{x \rightarrow \infty} \frac{1}{x} = 0$$

The way we read this is “as x approaches infinity, the function $1/x$ approaches 0 (but never reaches 0).” You will see this kind of “approach but never touch” behavior a lot especially when we dive into derivatives and integrals.

Using SymPy, we can calculate what value we approach for $f(x) = \frac{1}{x}$ as x approaches infinity ∞ .

Example 1-18. Using SymPy to calculate limits

```
from sympy import *

x = symbols('x')
f = 1 / x
result = limit(f, x, oo)

print(result) # 0
```

As you have seen, we discovered Euler’s number e this way too. It is the result of forever extending n into infinity for this function.

$$\lim_{n \rightarrow \infty} \left(1 + \frac{1}{n}\right)^n = e = 2.71828169413\dots$$

Funny enough, when we calculate Euler’s number with limits in SymPy (shown in [Example 1-19](#)), SymPy immediately recognizes it as Euler’s Number. We can call `evalf()` so we can actually display it as a number.

Example 1-19.

```
from sympy import *

n = symbols('n')
f = (1 + (1/n))**n
result = limit(f, n, oo)
```

```
print(result) # E  
print(result.evalf()) # 2.71828182845905
```

THE POWER OF SYMPY

SymPy is a powerful and fantastic computer algebra system (CAS) for Python that uses exact symbolic computation rather than approximate computation using decimals. It's helpful for those situations you would use "pencil and paper" to solve math and Calculus problems, with the benefit of using familiar Python syntax. Rather than represent the square root of 2 by approximating 1.4142135623730951, it will preserve it as exactly *sqrt(2)*.

So why not use SymPy for everything math-related? While we will use it throughout this book, it is important to still be comfortable doing Python math with plain decimals, as SciKit-Learn and other data science libraries take this approach. It is much faster for computers to use decimals rather than symbols. But keep SymPy in your back pocket as your advantage, and do not tell your high school and college-age kids about it. They can literally use it to do their math homework.

<https://docs.sympy.org/latest/tutorial/intro.html>

Derivatives

Let's go back to talking about functions and look at them from a Calculus perspective, starting with derivatives. A **derivative** tells the slope of a function, and it is useful to measure the rate of change at any point in a function.

Why do we care about derivatives? They are often used in machine learning and other mathematical algorithms, especially with gradient descent. When the slope is 0, that means we are at the minimum or maximum of an output variable. This concept will be useful later when we do linear regression, logistic regression, and neural networks.

But let's start with a simple example. A derivative provides the slope at that given x value. Let's take a look at the function $f(x) = x^2$ in [Figure 1-6](#). How “steep” is the curve at $x = 2$?

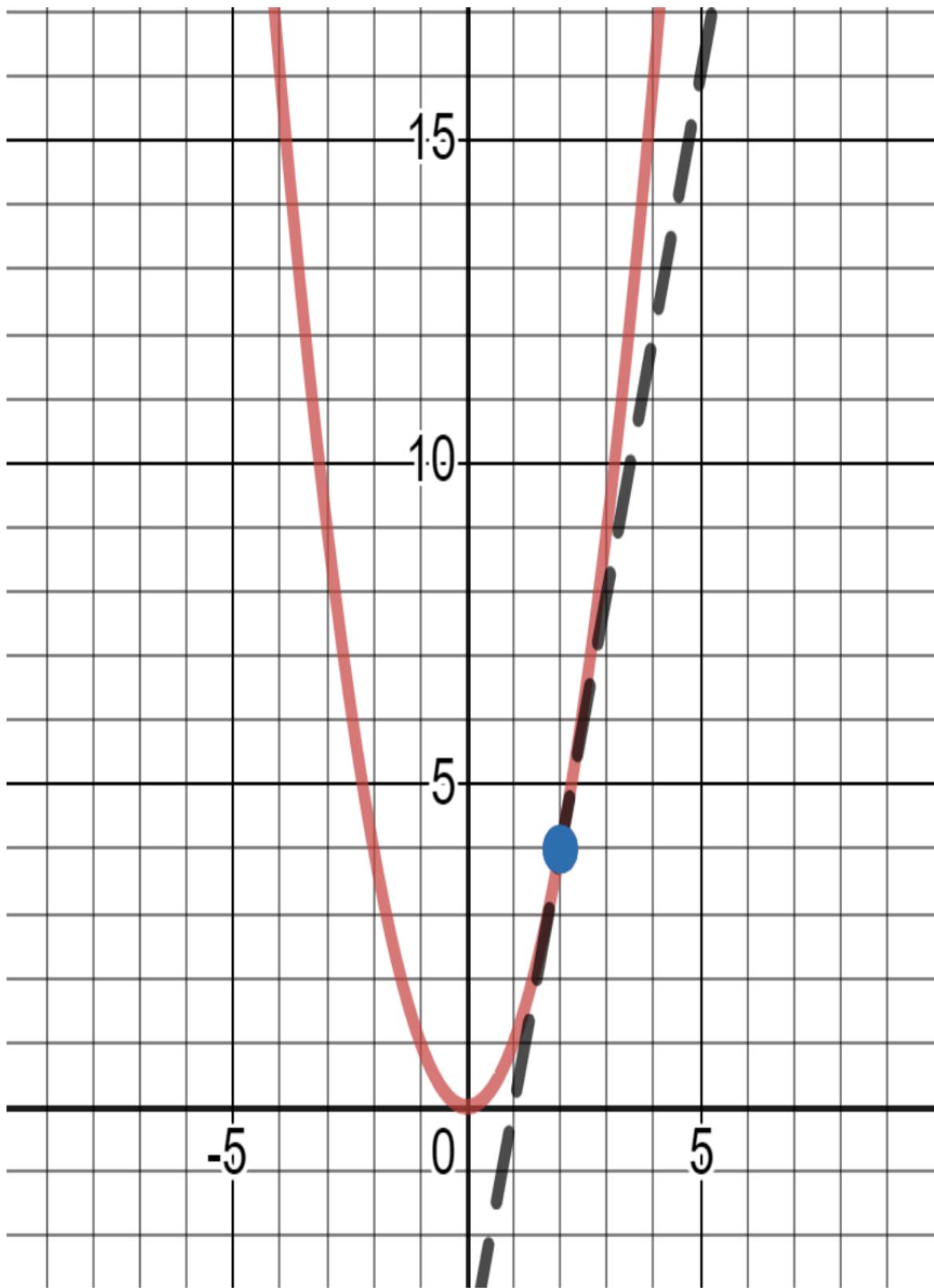


Figure 1-6. Observing steepness at a given part of the function

Notice above we can measure “steepness” at any point in the curve, and we can visualize this with a tangent line. Think of a **tangent line** as a straight line that “just touches” the curve at a given point. It also provides the slope at a given point. You can crudely estimate a tangent line at a given x value by creating a line intersecting that x value and a *really close* neighboring x value on the function.

Take $x = 2$ and a nearby value $x = 2.1$, which when passed to the function $f(x) = x^2$ will yield $f(2) = 4$ and $f(2.1) = 4.41$ as shown in [Figure 1-7](#). The resulting line that passes through these two points has a slope of 4.41.

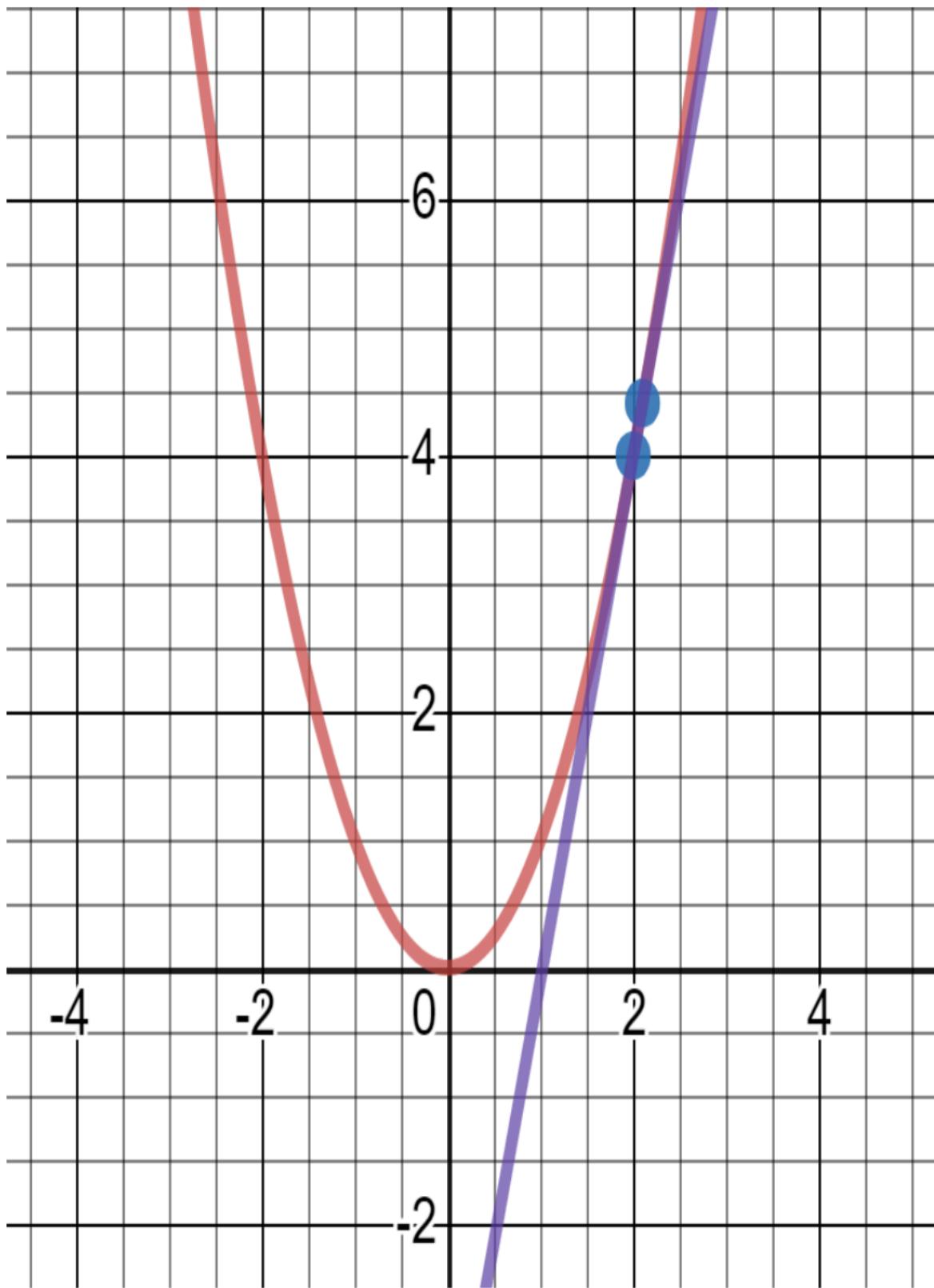


Figure 1-7. A crude way of calculating slope

You can quickly calculate the slope m between two points using the simple rise-over-run formula:

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

$$m = \frac{4.41 - 4.0}{2.1 - 2.0}$$

$$m = 4.41$$

If I made the x step between the two points even smaller, like $x = 2$ and $x = 2.00001$ which would result in $f(2) = 4$ and $f(2.00001) = 4.00004$, that would get *really* close to the actual slope of 4. So the smaller our step is to the neighboring value, the closer we get to the slope value at a given point in the curve. Like so many important concepts in math, we find something meaningful as we approach infinitely large or infinitely small values.

Example 1-20 shows a derivative calculator implemented in Python.

Example 1-20. A derivative calculator in Python

```
def derivative_x(f, x, step_size):
    m = (f(x + step_size) - f(x)) / ((x + step_size) - x)
    return m

def my_function(x):
    return x**2

slope_at_2 = derivative_x(my_function, 2, .00001)

print(slope_at_2) # prints 4.000010000000827
```

Now the good news is there is a cleaner way to calculate the slope anywhere on a function. We have already been using SymPy to plot graphs, but I will show you how it can also do tasks like derivatives for you using the magic of symbolic computation.

When you encounter an exponential function like $f(x) = x^2$ the derivative function will make the exponent a multiplier and then decrement the exponent by 1, leaving us with the derivative $\frac{d}{dx}x^2 = 2x$. The $\frac{d}{dx}$ indicates a

derivative with respect to x, which says we are building a derivative targeting the x value to get its slope. So if we want to find the slope at $x = 2$, and we have the derivative function, we just plug in that x value to get the slope.

$$f(x) = x^2$$

$$\frac{d}{dx} f(x) = \frac{d}{dx} x^2 = 2x$$

$$\frac{d}{dx} f(2) = 2(2) = 4$$

If you intend on learning these rules to hand-calculate derivatives, there are plenty of Calculus books for that. But there are some nice tools to calculate derivatives symbolically for you. The Python library SymPy is free and open-source, and it nicely adapts to using the Python syntax. [Example 1-21](#) shows how we calculate the derivative for $f(x) = x^2$ on SymPy.

Example 1-21. Calculating a derivative in SymPy

```
from sympy import *

# Declare 'x' to SymPy
x = symbols('x')

# Now just use Python syntax to declare function
f = x**2

# Calculate the derivative of the function
dx_f = diff(f)
print(dx_f) # prints 2*x
```

Wow! So by declaring variables using the `symbols()` function in SymPy, I can then proceed to use normal Python syntax to declare my function. After that I can use `diff()` to calculate the derivative function for me. In [Example 1-22](#) can then take our derivative function back to plain Python and simply declare it as another function.

Example 1-22. A derivative calculator in Python

```

def f(x):
    return x**2

def dx_f(x):
    return 2*x

slope_at_2 = dx_f(2.0)

print(slope_at_2) # prints 4.0

```

If you wanted to keep using SymPy, you can call the `subs()` function to swap the x variable with the value 2 as shown in [Example 1-23](#).

Example 1-23. Using the substitution feature in SymPy

```

# Calculate the slope at x = 2
print(dx_f.subs(x,2)) # prints 4

```

Another concept we will encounter in this book is **partial derivatives** in [Chapter 5](#), Chapter 6, and Chapter 7, which are derivatives on functions that have multiple input variables. Think of it this way. Rather than finding the slope on a 1-dimensional function, we have slopes with respect to multiple variables in several directions. For each given variable derivative, we assume the other variables are held constant. Take a look at the 3D graph of $f(x, y) = 2x^3 + 3y^3$ in [Figure 1-8](#) and you will see we have slopes in two directions for two variables.

Let's take the function $f(x, y) = 2x^3 + 3y^3$. The x and y variable each get their own derivatives $\frac{d}{dx}$ and $\frac{d}{dy}$. These represent the slope values with respect to each variable on a multidimensional surface. These are the derivatives for x and y , followed by the SymPy code to calculate those derivatives:

$$f(x, y) = 2x^3 + 3y^3$$

$$\frac{d}{dx} 2x^3 + 3y^3 = 4x$$

$$\frac{d}{dy} 2x^3 + 3y^3 = 9y^2$$

Example 1-24 and **Figure 1-8** shows how we calculate the partial derivatives for x and y respectively with SymPy.

Example 1-24. Calculating partial derivatives with SymPy

```
from sympy import *
from sympy.plotting import plot3d

# Declare x and y to SymPy
x,y = symbols('x y')

# Now just use Python syntax to declare function
f = 2*x**3 + 3*y**3

# Calculate the partial derivatives for x and y
dx_f = diff(f, x)
dy_f = diff(f, y)

print(dx_f) # prints 6*x**2
print(dy_f) # prints 9*y**2

# plot the function
plot3d(f)
```

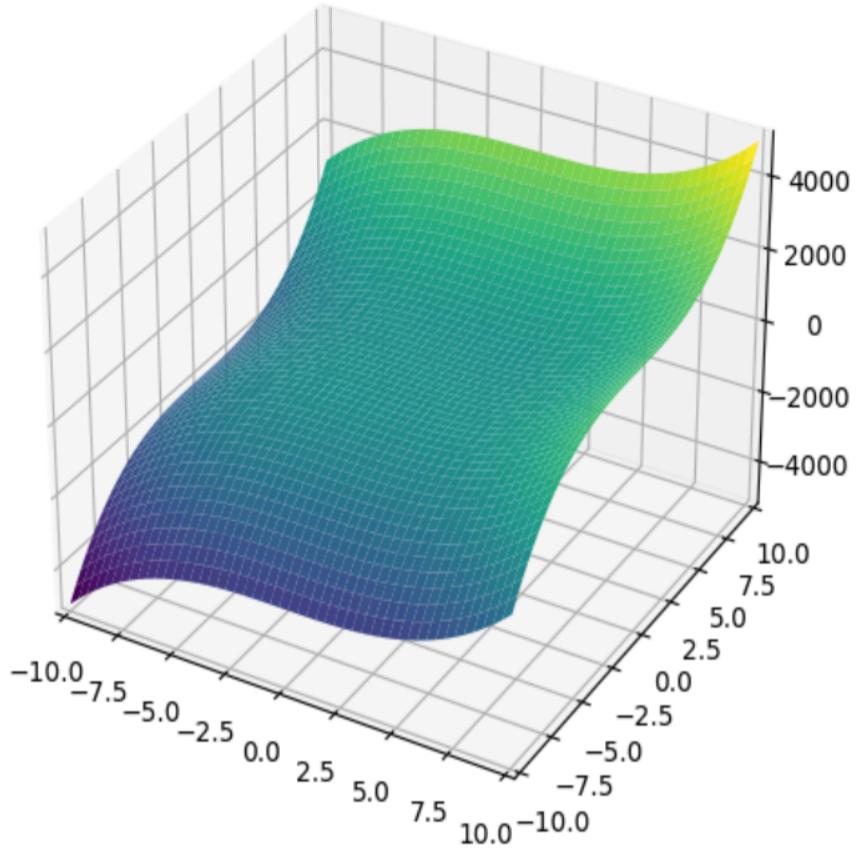


Figure 1-8. Plotting a 3-dimensional exponential function

So for (x,y) values $(1,2)$, the slope with respect to x is $6(1) = 4$ and the slope with respect to y is $9(2)^2 = 36$.

USING LIMITS TO CALCULATE DERIVATIVES

Want to see where limits come into play calculating derivatives? If you are feeling good about what we learned so far, proceed! If you are still digesting, maybe consider coming back to this sidebar later.

Sympy allows us to do some interesting explorations about math. Take our function $f(x) = x^2$ and we approximated a slope for $x = 2$ by drawing a line through a close neighboring point $x = 2.0001$ by adding a step $.0001$. Why not use a limit to forever decrease that step s and see what slope it approaches?

$$\lim_{s \rightarrow 0} \frac{(x + s)^2 - x^2}{(x + s) - x}$$

In our example, we are interested in the slope where $x = 2$ so let's substitute that.

$$\lim_{s \rightarrow 0} \frac{(2 + s)^2 - 2^2}{(2 + s) - 2} = 4$$

By forever approaching a step size s to 0 but never reaching it (remember the neighboring point cannot touch the point at $x = 2$ else we have no line!), we can use a limit to see we converge on a slope of 4 as shown in [Example 1-25](#).

Example 1-25. Using limits to calculate a slope

```
from sympy import *

# "x" and step size "s"
x, s = symbols('x s')

# declare function
f = x**2

# slope between two points with gap "s"
# substitute into rise-over-run formula
slope_f = (f.subs(x, x + s) - f) / ((x+s) - x)
```

```

# substitute 2 for x
slope_2 = slope_f.subs(x, 2)

# calculate slope at x = 2
# infinitely approach step size _s_ to 0
result = limit(slope_2, s, 0)

print(result) # 4

```

Now what if we do not assign a specific value to x and leave it alone? What happens if we decrease our step size s infinitely towards 0 then? Let's look in [Example 1-26](#).

Example 1-26. Using limits to calculate a derivative

```

from sympy import *

# "x" and step size "s"
x, s = symbols('x s')

# declare function
f = x**2

# slope between two points with gap "s"
# substitute into rise-over-run formula
slope_f = (f.subs(x, x + s) - f) / ((x+s) - x)

# calculate derivative function
# infinitely approach step size +s+ to 0
result = limit(slope_f, s, 0)

print(result) # 2x

```

Wow! That gave us our derivative function $2x$. SymPy was smart enough to figure out to never let our step size reach 0 but forever approach 0. This converges $f(x) = x^2$ to reach its derivative counterpart $2x$.

Integrals

The opposite of a derivative is an **integral**, which finds the area under the curve for a given range. We will find ourselves using integrals quite a bit in this book to find areas under probability distributions.

I want to take an intuitive approach for learning integrals called the Riemann Sums, one that flexibly adapts to any continuous function.

First, let's point out that finding the area for a range under a straight line is easy. Let's say I have a function $f(x) = 2x$ and I want to find the area under the line between 0 and 1, as shaded in [Figure 1-9](#).

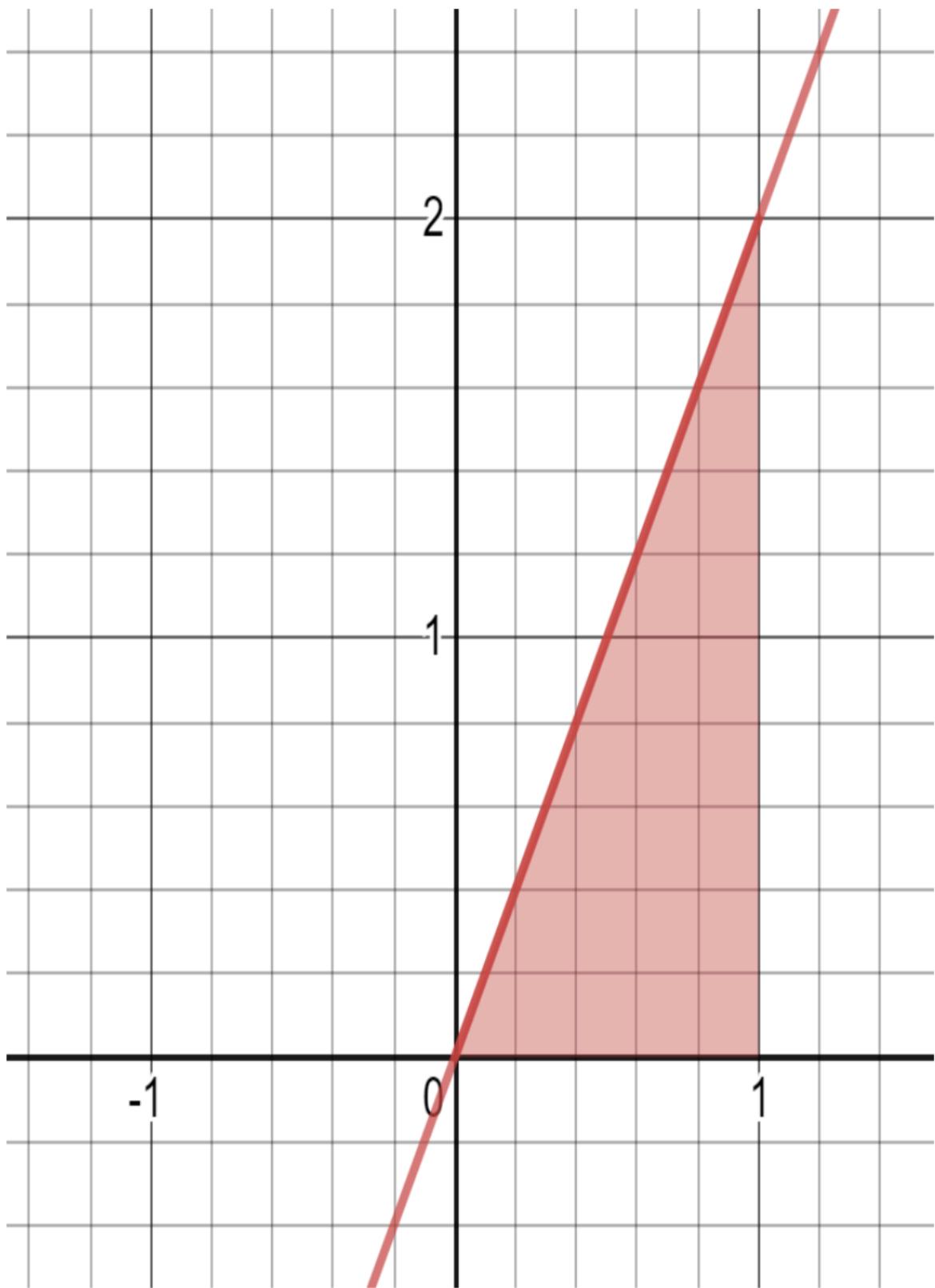


Figure 1-9. Calculating an area under a linear function

Notice that I am finding the area bounded between the line and the x-axis, and in the x range 0 to 1.0. If you recall basic geometry formulas, the area A for a triangle is $A = \frac{1}{2}b*h$ where b is the length of the base and h is the height.

We can visually spot above that $b = 1$ and $h = 2$. So plugging into the formula we get for our area 1.0 as calculated below:

$$A = \frac{1}{2}bh$$

$$A = \frac{1}{2}*1.0*2.0$$

$$A = 1.0$$

That was not bad, right? But let's look at a function that is difficult to find the area under: $f(x) = x^2 + 1$. What is the area between 0 and 1 as shaded in [Figure 1-10](#)?

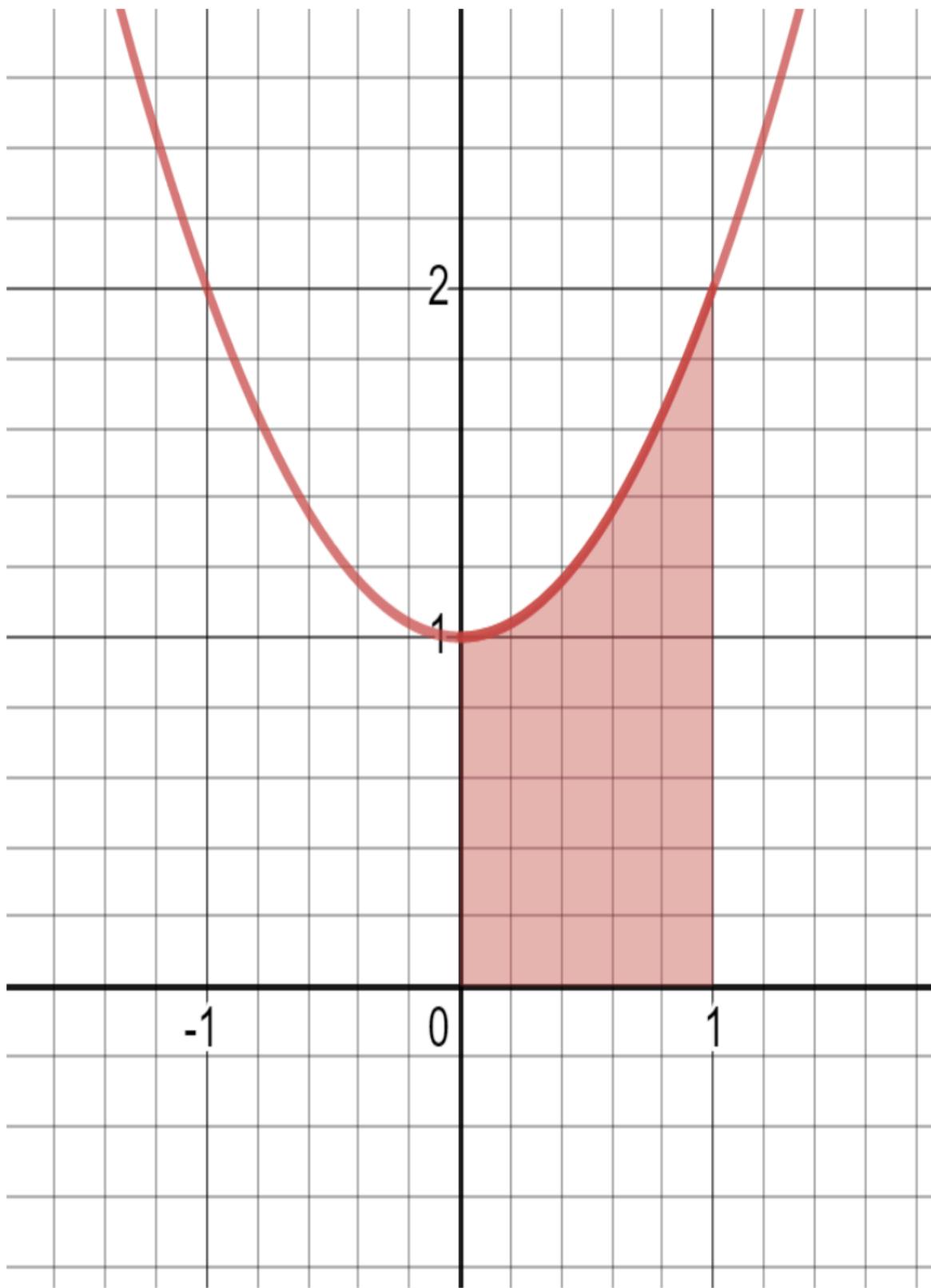


Figure 1-10. Calculating area under nonlinear functions is less straightforward

Again we are only interested in the area below the curve and above the x-axis, only within the x range between 0 and 1. The curviness here does not give us a clean geometric formula to find the area, but here is a clever little hack you can do.

What if we packed 5 rectangles of equal length under the curve as shown in [Figure 1-11](#), where the height of each one extends from the x-axis to where the midpoint touches the curve?

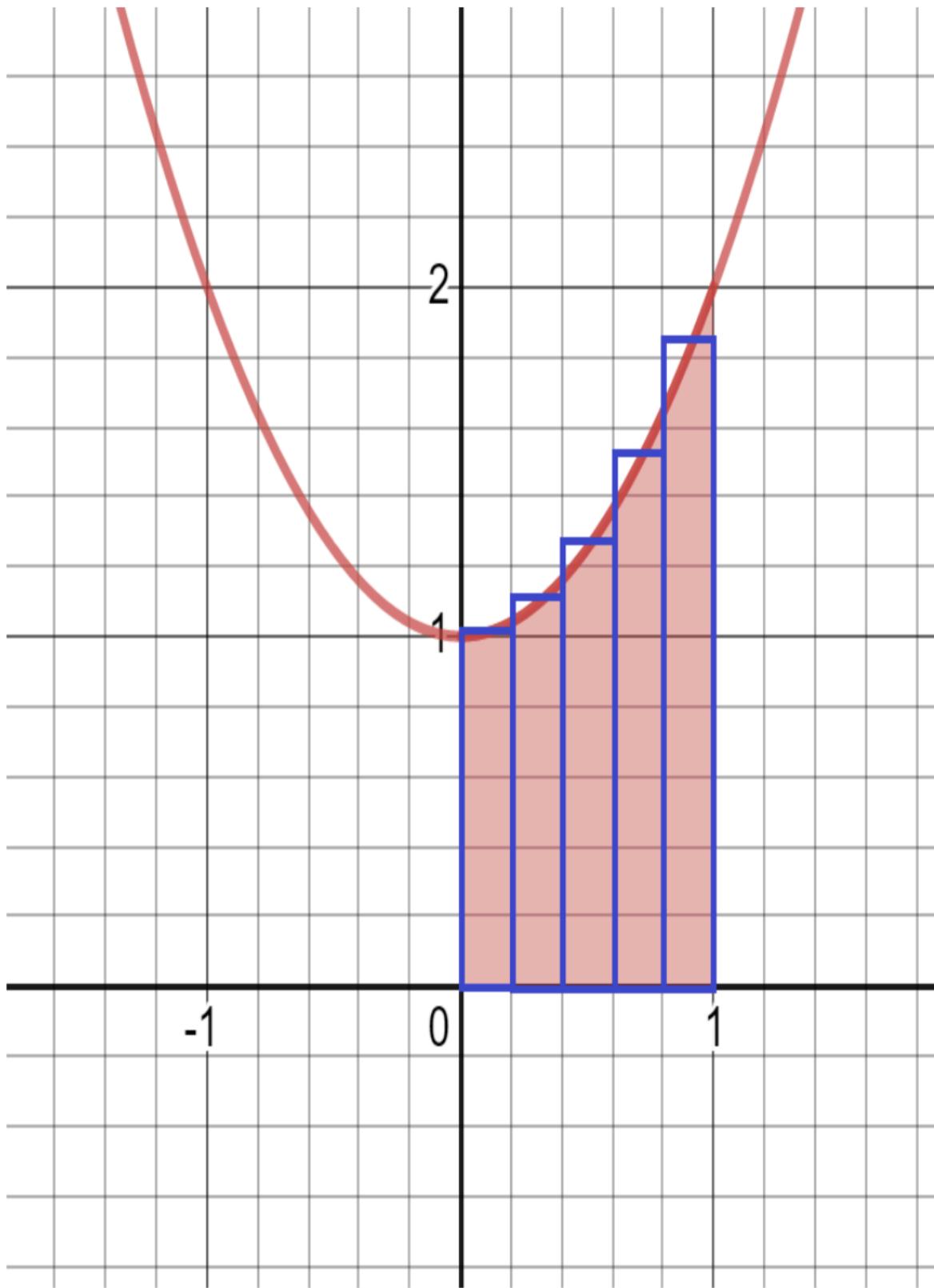


Figure 1-11. Packing rectangles under a curve to approximate area

The area of a rectangle is $A = \text{length} * \text{width}$, so we could easily sum the areas of the rectangles. Would that give us a good approximation of the area under the curve? What if we packed 100 rectangles? 1000? 100,000? As we increase the number of rectangles while decreasing their width, would we not get closer to the area under the curve? Yes we would, and it is yet another case where we increase/decrease something towards infinity to approach an actual value.

Let's try it out in Python! First we need a function that approximates an integral which we will call `approximate_integral()`. The arguments a and b will specify the min and max of the x range respectively. n will be the number of rectangles to pack, and f will be the function we are integrating. We implement the function in [Example 1-27](#), and then use it to integrate our function $f(x) = x^2 + 1$ with 5 rectangles, between 0.0 and 1.0.

Example 1-27. An integral approximation in Python

```
def approximate_integral(a, b, n, f):
    delta_x = (b - a) / n
    total_sum = 0

    for i in range(1, n + 1):
        midpoint = 0.5 * (2 * a + delta_x * (2 * i - 1))
        total_sum += f(midpoint)

    return total_sum * delta_x

def my_function(x):
    return x**2 + 1

area = approximate_integral(a=0, b=1, n=5, f=my_function)

print(area) # prints 1.33
```

So we get an area of 1.33. What happens if we use 1000 rectangles? Let's try it out in [Example 1-28](#).

Example 1-28. An integral approximation in Python

```
area = approximate_integral(a=0, b=1, n=1000, f=my_function)

print(area) # prints 1.333333250000001
```

Okay we are getting some more precision here, and getting some more decimal places. What about 1 million rectangles as shown in [Example 1-29](#)?

Example 1-29. An integral approximation in Python

```
area = approximate_integral(a=0, b=1, n=1_000_000, f=my_function)

print(area) # prints 1.333333333332733
```

Okay I think we are getting a diminishing return here, and are converging on the value 1.333 where the “.333” part is forever recurring. If this were a rational number, it is likely $\frac{4}{3} = 1.333$. As we increase the number of rectangles, the approximation starts to reach its limit at smaller and smaller decimals.

Now that we got some intuition on what we are trying to achieve and why, let’s do a more exact approach with SymPy which happens to support rational numbers in [Example 1-30](#).

Example 1-30. Using SymPy to perform integration

```
from sympy import *

# Declare 'x' to SymPy
x = symbols('x')

# Now just use Python syntax to declare function
f = x**2 + 1

# Calculate the integral of the function with respect to x
# for the area between x = 0 and 1
area = integrate(f, (x, 0, 1))

print(area) # prints 4/3
```

Cool! So the area actually is $4/3$, which is what our previous method converged on. Unfortunately plain Python (and many programming languages) only support decimals but computer algebra systems like SymPy give us exact rational numbers. We will be using integrals to find areas under curves in [Chapter 2](#) and [Chapter 3](#), although we will have Scikit-Learn do the work for us when available.

CALCULATING INTEGRALS WITH LIMITS

For the super curious, here is how we calculate definite integrals using limits in SymPy. Please! Skip or come back to this sidebar later if you have enough to digest. But if you are feeling good and want to deep dive how integrals are derived using limits, proceed!

The main idea follows much what we did earlier: pack rectangles under a curve and make them infinitely smaller until we approach the exact area. But of course, the rectangles cannot have a width of 0... they just have to keep getting closer to 0 without ever reaching 0. It is another case of using limits.

Khan Academy has a great article explaining how to use limits for Reimann Sums but here is how we do it in SymPy as shown in [Example 1-31](#).

Example 1-31. Using Limits to Calculate Integrals

```
from sympy import *

# Declare variables to SymPy
x, i, n = symbols('x i n')

# Declare function and range
f = x**2 + 1
lower, upper = 0, 1

# Calculate width and each rectangle height at index "i"
delta_x = ((upper - lower) / n)
x_i = (lower + delta_x * i)
fx_i = f.subs(x, x_i)

# Iterate all "n" rectangles and sum their areas
n_rectangles = Sum(delta_x * fx_i, (i, 1, n)).doit()

# Calculate the area by approaching the number
# of rectangles "n" to infinity
area = limit(n_rectangles, n, oo)

print(area) # prints 4/3
```

Above we determine the length of each rectangle `delta_x_` and the start of each rectangle `+x_i_` where `i` is the index of each rectangle. `+fx_i_` is the height of each rectangle at index `i`. We declare `n` number of rectangles and sum their areas `+delta_x * fx_i_`, but we have no area value yet because we have not committed a number for `n`. Instead we approach `n` towards infinity to see what area we converge on, and you should get `+4/3!`

Conclusion

In this chapter we covered some foundations we will use for the rest of this book. From number theory to logarithms and calculus integrals, we cherrypicked some important mathematical concepts that have relevance in data science, machine learning, and analytics. You may have questions about why these concepts are useful. That will come next!

Before we move on to discuss probability, take a little bit of time to skim these concepts one more time and then do the following exercise. You can always revisit this chapter as you progress through this book, and refresh as necessary when you start applying these mathematical ideas.

Exercises

1. Is the value 62.6738 rational or irrational? Why or why not?
2. Evaluate the expression: $10^7 10^{-5}$
3. Evaluate the expression: $81^{\frac{1}{2}}$
4. Evaluate the expression: $25^{\frac{3}{2}}$
5. Assuming no payments are made, how much would a \$1000 loan be worth at 5% interest compounded monthly after 3 years?

6. Assuming no payments are made, how much would a \$1000 loan be worth at 5% interest compounded continuously after 3 years?
7. For the function $f(x) = 3x^2 + 1$ what is the slope at $x = 3$?
8. For the function $f(x) = 3x^2 + 1$ what is the area under the curve for x between 0 and 2?

Chapter 2. Probability

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at thomasnield@live.com.

When you think of probability, what images come to mind? Perhaps you think of gambling-related examples, like the probability of winning the lottery or getting a pair with two dice. Maybe it is predicting stock performance, the outcome of a political election, or whether your flight will arrive on time. Our world is full of uncertainties we want to measure.

Maybe that is the word we should focus on: uncertainty. How do we measure something that we are uncertain about?

In the end, probability is the theoretical study of measuring certainty that an event will happen. It is a foundational discipline for statistics, hypothesis testing, machine learning, and other topics in this book. A lot of folks take probability for granted and assume they understand it. However, it is more nuanced and complicated than most people think. While the theorems and ideas of probability are mathematically sound, it gets more complex when we introduce data and venture into statistics. We will cover that in [Chapter 4](#) and hypothesis testing.

In this chapter, we will discuss what probability is. Then we will cover probability math concepts, Bayes Theorem, the Binomial Distribution, and the Beta Distribution.

Understanding Probability

Probability is how strongly we believe an event will happen, often expressed as a percentage. Here are some questions that might warrant a probability for an answer:

- How likely will I get 7 heads in 10 fair coin flips?
- What are my chances in winning an election?
- Will my flight be late?
- How certain am I a product is defective?

The most popular way to express probability is as a percentage, as in “There is a 70% chance my flight will be late.” We will call this probability $P(X)$, where X is the event of interest. As you work with probabilities though, you will more likely see it expressed as a decimal (in this case .70) which must be between 0.0 and 1.0.

$$P(X) = .70$$

Likelihood is similar to probability, and it is easy to confuse the two (many dictionaries do as well). You can get away with using “probability” and “likelihood” interchangeably in everyday conversation. However, we should pin down these differences. Probability is about quantifying predictions of events yet to happen, whereas likelihood is measuring the frequency of events that already occurred. In statistics and machine learning, we often use likelihood (the past) in the form of data to predict probability (the future).

It is important to note that a probability of an event happening must be strictly between 0% and 100%, or 0.0 and 1.0. Logically, this means the probability of an event *NOT* happening is calculated by subtracting the probability of the event from 1.0.

$$P(X) = .70$$

$$P(\text{not } X) = 1.0 - .70 = .30$$

Alternatively, probability can be expressed as an **odds** $O(X)$ expressed as 7:3, $\frac{7}{3}$, or 2.333.

ODDS ARE USEFUL!

While many people feel more comfortable expressing probabilities as percentages or proportions, odds can be a helpful tool. If I have an odds of 2.0, that means I feel an event is 2 times more likely to happen than not to happen. That can be more intuitive to describe a belief than a percentage of 66.666%. For this reason, odds are helpful quantifying subjective beliefs especially in a gambling/betting context. It plays a role in Bayesian statistics (including the Bayes Factor) as well as logistic regressions with the log-odds which we will cover in Chapter 6.

To turn an odds $O(X)$ into a proportional probability $P(X)$, use the formula below:

$$P(X) = \frac{O(X)}{1.0 + O(X)}$$

So if have an odds $\frac{7}{3}$, I can convert it into a proportional probability like this:

$$P(X) = \frac{O(X)}{1.0 + O(X)}$$

$$P(X) = \frac{\frac{7}{3}}{1.0 + \frac{7}{3}}$$

$$P(X) = .7$$

Conversely, you can turn an odds into a probability simply by dividing the probability of the event occurring by the probability it will not occur:

$$O(X) = \frac{P(X)}{1.0 - P(X)}$$

$$O(X) = \frac{.70}{1.0 - .70}$$

$$O(X) = \frac{7}{3}$$

Probability versus Statistics

Sometimes people use the terms *probability* and *statistics* interchangeably, and while this is understandable to conflate the two disciplines they do have distinctions.

Probability is purely theoretical of how likely an event is to happen, and does not require data. **Statistics** on the other hand cannot exist without data, and uses it to discover probability and provides tools to describe data.

Think of predicting the outcome of rolling a “4” on a die (that’s the singular of “dice”). Approaching the problem with a pure probability mindset, one simply says there are 6 sides on a die. We assume each side is equally likely, so the probability of getting a “4” is $\frac{1}{6}$, or 16.666%.

However, a zealous statistician might say “No! We need to roll the die to get data. If we can get 30 rolls or more, and the more rolls we do the better, only then will we have data to determine the probability of getting a 4.” This approach may seem silly if we assume the die is fair; but what if it’s not? If that’s the case, collecting data is the only way to discover the probability of rolling a “4”. We will talk about hypothesis testing in [Chapter 3](#).

BAYESIAN AND FREQUENTIST STATISTICS

If you are curious to learn the differences between Bayesian and Frequentist thinking, turn to Appendix A.

Probability Math

When we work with a single probability of an event $P(X)$, known as a **marginal probability**, the idea is fairly straightforward as discussed previously. But when we start combining probabilities of different events together, it gets a little less intuitive.

Joint Probabilities

Let’s say you have a fair coin and a fair 6-sided die. You want to find the probability of flipping a “Heads” and rolling a “6” on the coin and die respectively. These are two separate probabilities of two separate events, but we want to find the probability that both events will occur together. This is known as a **joint probability**.

Think of a joint probability as an AND operator. I want to find the probability of flipping a *heads* AND rolling a *six*. We want both events to happen together, so how do we calculate this probability?

There are 2 sides on a coin and 6 sides on the die, so the probability of heads is $\frac{1}{2}$ and probability of six is $\frac{1}{6}$. The probability of both events occurring (assuming they are independent, more on this later!) is simply multiplying the two together:

$$P(A \text{ AND } B) = P(A)*P(B)$$

$$P(\text{Heads}) = \frac{1}{2}$$

$$P(6) = \frac{1}{6}$$

$$P(\text{Heads AND 6}) = \frac{1}{2} * \frac{1}{6} = \frac{1}{12} = .08333$$

Easy enough, but why is this the case? A lot of probability rules can be discovered by generating all possible combinations of events, which comes from an area of math known as permutations and combinations. For this case, generate every possible outcome between the coin and die, pairing heads (H) and tails (T) with the numbers 1 through 6. Note I put asterisks “**” around the outcome of interest where we got heads and a six.

H1 H2 H3 H4 H5 *H6* T1 T2 T3 T4 T5 T6

Notice there are 12 possible outcomes when flipping our coin and rolling our die. The only one that is of interest to us is “H6,” getting a heads and a six. So because there is only 1 outcome that satisfies our condition, and there are 12 possible outcomes, the probability of getting a heads and a six is $\frac{1}{12}$.

Rather than generate all possible combinations and counting the ones of interest to us, we can again use the multiplication as a shortcut to find the joint probability. This is known as the **product rule**.

$$P(A \text{ AND } B) = P(A)*P(B)$$

$$P(\text{Heads AND 6}) = \frac{1}{2} * \frac{1}{6} = \frac{1}{12} = .08333$$

Union Probabilities

We discussed joint probabilities, which is the probability of two or more events occurring simultaneously. But what about the probability of getting event A or B?

When we deal with “OR” operations with probabilities, this is known as a **union probability**.

Let’s start with **mutually exclusive** events, which are events that cannot occur simultaneously. For example, if I roll a die I cannot simultaneously get a “4” and a “6”. I can only get one outcome. Getting the union probability for these cases are easy. I just simply add them together. If I want to find the probability of getting a “4” or “6” on a die roll, it is going to be $\frac{2}{6} = \frac{1}{3}$.

$$P(4) = \frac{1}{6}$$

$$P(6) = \frac{1}{6}$$

$$P(4 \text{ OR } 6) = \frac{1}{6} + \frac{1}{6} = \frac{1}{3}$$

But what about **non-mutually exclusive** events, which are events that can occur simultaneously? Let’s go back to the coin flip and die roll example. What is the probability of getting a heads OR a “six”? Before you are tempted to add those probabilities, let’s generate all possible outcomes again and highlight the ones we are interested in:

H1 *H2* *H3* *H4* *H5* *H6* T1 T2 T3 T4 T5 *T6*

Above we are interested in all the “heads” outcome as well as the “6” outcomes. If we proportion the 7 out of 12 outcomes we are interested in, $\frac{7}{12}$, we get a correct probability of .58333.

But what happens if we add the probabilities of heads and “6” together? We get a different (and wrong!) answer of .666.

$$P(HEADS) = \frac{1}{2}$$

$$P(6) = \frac{1}{6}$$

$$P(HEADS \text{ OR } 6) = \frac{1}{2} + \frac{1}{6} = \frac{4}{6} = .666$$

Why is that? Study the combinations of coin flip and die outcomes again and see if you can find something fishy. Notice when we add the probabilities, we double-count the probability of getting a “6” in both “H6” and “T6”! If this is not clear, try finding the probability of getting “Heads” and a die roll of 1 through 5.

$$P(\text{HEADS}) = \frac{1}{2}$$

$$P(1 \text{ through } 5) = \frac{5}{6}$$

$$P(\text{HEADS OR 1 through 5}) = \frac{1}{2} + \frac{5}{6} = \frac{8}{6} = 1.333$$

We get a probability of 133.333% which is definitely not correct, as a probability must be no more than 100% or 1.0. The problem again is we are double-counting outcomes.

If you ponder long enough, you may realize the logical way to remove double-counting in a union probability is to subtract the joint probability. This is known as the **sum rule of probability** and ensures every joint event is only counted once.

$$P(A \text{ OR } B) = P(A) + P(B) - P(A \text{ AND } B)$$

$$P(A \text{ OR } B) = P(A) + P(B) - P(A)*P(B)$$

So going back to our example of calculating the probability of a Heads or a “six”, we need to subtract the joint probability of getting a Heads or a “six” from the union probability:

$$P(\text{HEADS}) = \frac{1}{2}$$

$$P(6) = \frac{1}{6}$$

$$P(A \text{ OR } B) = P(A) + P(B) - P(A)*P(B)$$

$$P(\text{HEADS OR 6}) = \frac{1}{2} + \frac{1}{6} - \left(\frac{1}{2} * \frac{1}{6} \right) = .58333$$

Note that this formula also applies to mutually exclusive events. If the events are mutually exclusive where only one outcome A or B is allowed but not both, then the

joint probability $P(A \text{ AND } B)$ is going to be 0, and therefore remove itself from the formula. You are then left with simply summing the events as we did earlier.

In summary, when you have a union probability between two or more events that are not mutually exclusive, be sure to subtract the joint probability so no probabilities are double-counted.

Conditional Probability and Bayes Theorem

A probability topic that people easily get confused by is the concept of **conditional probability**, which is the probability of an event A occurring given event B has occurred. It is typically expressed as $P(A \text{ GIVEN } B)$ or $P(A|B)$.

Let's say some study makes a claim that 85% of cancer patients drank coffee. How do you react to this claim? Does this alarm you and make you want to abandon your favorite morning drink? Let's first define this as a conditional probability $P(\text{Coffee given Cancer})$ or $P(\text{Coffee}|\text{Cancer})$. This represents a probability of people who drink coffee given they have cancer.

Within the United States, let's compare this to the percentage of people diagnosed with cancer (0.5% according to cancer.gov) and the percentage of people who drink coffee (65% according to statista.com):

$$P(\text{Coffee}) = .65$$

$$P(\text{Cancer}) = .005$$

$$P(\text{Coffee}|\text{Cancer}) = .85$$

Hmm... study these numbers for a moment and ask whether coffee is really contributing to cancer here. Notice again that only 0.5% of the population has cancer at any given time. However 65% of the population drinks coffee regularly. If coffee contributes to cancer, should we not have much higher cancer numbers than 0.5%? Would it not be closer to 65%?

This is the sneaky thing about proportional numbers. They may seem significant without any given context, and media headlines can certainly exploit this for clicks: "New Study Reveals 85% of Cancer Patients Drink Coffee" it might read. Of course, this is silly because we have taken a common attribute (drinking coffee) and associated it with an uncommon one (having cancer).

The reason people can be so easily confused by conditional probabilities is because the direction of the condition matters, and the two conditions are conflated as somehow

being equal. The “probability of having cancer given you are a coffee drinker” is different than the “probability of being a coffee drinker given you have cancer.” To put it simply: few coffee drinkers have cancer, but many cancer patients drink coffee.

If we are interested in studying whether coffee contributes to cancer, we really are interested in the first conditional probability: the probability of someone having cancer given they are a coffee drinker.

$$P(\text{Coffee}|\text{Cancer}) = .85$$

$$P(\text{Cancer}|\text{Coffee}) = ?$$

How do we flip the condition? There’s a powerful little formula called **Bayes Theorem** and we can use it to flip conditional probabilities.

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

If we plug the information we already have into this formula, we can solve for the probability someone has cancer given they drink coffee:

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

$$P(\text{Cancer}|\text{Coffee}) = \frac{P(\text{Coffee}|\text{Cancer})*P(\text{Coffee})}{P(\text{Cancer})}$$

$$P(\text{Cancer}|\text{Coffee}) = \frac{P(.85)*P(.005)}{P(.65)} = .0065$$

If you want to calculate this in Python, check out [Example 2-1](#).

Example 2-1. Using Bayes Theorem in Python

```
p_coffee_drinker = .65
p_cancer = .005
p_coffee_drinker_given_cancer = .85

p_cancer_given_coffee_drinker = p_coffee_drinker_given_cancer *
    p_cancer / p_coffee_drinker

# prints 0.006538461538461539
print(p_cancer_given_coffee_drinker)
```

Wow! So the probability someone has cancer given they are a coffee drinker is only 0.65%! This number is very different than the probability someone is a coffee drinker given they have cancer, which is 85%. Now do you see why the direction of the condition matters? Bayes Theorem is helpful for this reason, and it can be used to chain several conditional probabilities together to keep updating our beliefs based on new information.

WHAT DEFINES A “COFFEE DRINKER”?

Note that I could have accounted for other variables here, in particular what qualifies someone as a “coffee drinker.” If someone drinks coffee once a month, as opposed to someone who drinks coffee every day, should I equally qualify both as “coffee drinkers”? What about the person who started drinking coffee a month ago as opposed to someone who drank coffee for 20 years? How often and how long do people have to drink coffee before they meet the threshold of being a “coffee drinker” in this cancer study?

These are important questions to consider, and shows why data rarely tells a whole story. If someone gives you a spreadsheet of patients with a simple “YES/NO” flag on whether they are a coffee drinker, that threshold needs to be defined! Or we need a more weightful metric like “number of coffee drinks consumed in the last 3 years.” I kept this thought experiment simple and didn’t define how someone qualifies as a “coffee drinker”, but be aware that out in the field, always ask for context about the data.

If you want to explore the intuition behind Bayes Theorem more deeply, turn to Appendix A. For now just know it helps us flip a conditional probability. Let’s talk about how conditional probabilities interact with joint and union operations next.

NAIVE BAYES

Bayes Theorem plays a central role to a common machine learning algorithm called Naive Bayes. Joel Grus covers it in his book Data Science from Scratch [<https://www.oreilly.com/library/view/data-science-from/9781492041122/>].

Joint and Union Conditional Probabilities

Let’s revisit joint probabilities and how they interact with conditional probabilities. I want to find the probability somebody is a coffee drinker AND they have cancer.

Should I multiply $P(\text{Coffee})$ and $P(\text{Cancer})$? Or should I use $P(\text{Coffee}|\text{Cancer})$ in place of $P(\text{Coffee})$ if it is available? Which one do I use?

OPTION 1:

$$P(\text{Coffee}) \times P(\text{Cancer}) = .65 \times .005 = .00325$$

OPTION 2:

$$P(\text{Coffee}|\text{Cancer}) \times P(\text{Cancer}) = .85 \times .005 = .00425$$

If we already have established our probability only applies to people with Cancer, does it not make sense to use $P(\text{Coffee}|\text{Cancer})$ instead of $P(\text{Coffee})$? One is more specific and applies to a condition that's already been established. So we should use $P(\text{Coffee}|\text{Cancer})$ as $P(\text{Cancer})$ is already part of our joint probability. This means the probability of someone having cancer and being a coffee drinker is 0.425%.

$$P(\text{Coffee and Cancer}) = P(\text{Coffee}|\text{Cancer}) \times P(\text{Cancer}) = .85 \times .005 = .00425$$

This joint probability also applies in the other direction. I can find the probability of someone being a coffee drinker and having cancer by multiplying $P(\text{Cancer}|\text{Coffee})$ and $P(\text{Coffee})$. As you can observe, I arrive at the same answer:

$$P(\text{Cancer}|\text{Coffee}) \times P(\text{Coffee}) = .0065 \times .65 = .00425$$

If we did not have any conditional probabilities available, then the best we can do is multiply $P(\text{Coffee Drinker})$ and $P(\text{Cancer})$ as shown here:

$$P(\text{Coffee Drinker}) \times P(\text{Cancer}) = .65 \times .005 = .00325$$

Now think about this: if event A has no impact on event B, then what does that mean for conditional probability $P(B|A)$? That means $P(B|A) = P(B)$, meaning event A occurring makes no difference to how likely event B is to occur. Therefore we can update our joint probability formula to be, regardless if the two events are dependent, to be:

$$P(A \text{ AND } B) = P(B) \times P(A|B)$$

And finally let's talk about unions and conditional probability. If I wanted to calculate the probability of A or B occurring, but A may affect the probability of B, we update our sum rule like this:

$$P(A \text{ OR } B) = P(A) + P(B) - P(A|B) \times P(B)$$

As a reminder, this applies to mutually exclusive events as well. The sum rule $P(A|B)*P(B)$ would yield 0 if the events A and B cannot happen simultaneously.

Binomial Distribution

Let's say you are working on a new turbine jet engine and ran 10 tests. The outcomes yielded 8 successes and 2 failures:

✓ ✓ ✓ ✓ ✓ ✗ ✓ ✗ ✓ ✓

You were hoping to get a 90% success rate, but based on the data above you conclude that your tests have failed with only 80% success. Each test is time-consuming and expensive, so you decide it is time to go back to the drawing board to re-engineer the design.

However one of your engineers insists there should be more tests. "The only way we will know for sure is to run more tests," she argues. "What if more tests yield 90% or greater success? After all, if you flip a coin 10 times and get 8 heads, it does not mean the coin is fixed at 80%."

You briefly consider the engineer's argument and realize she has a point. Even a fair coin flip will not always have an equally split outcome, especially with only 10 flips. You are most likely to get 5 heads but you can also get 3, 4, 6, or 7 heads. You could even get 10 heads, although this is unlikely. So how do you determine the likelihood of 80% success assuming the underlying probability is 90%?

One tool that might be relevant here is the **binomial distribution**, which measures how likely k successes can happen out of n trials given p probability.

Visually, a binomial distribution looks like [Figure 2-1](#).

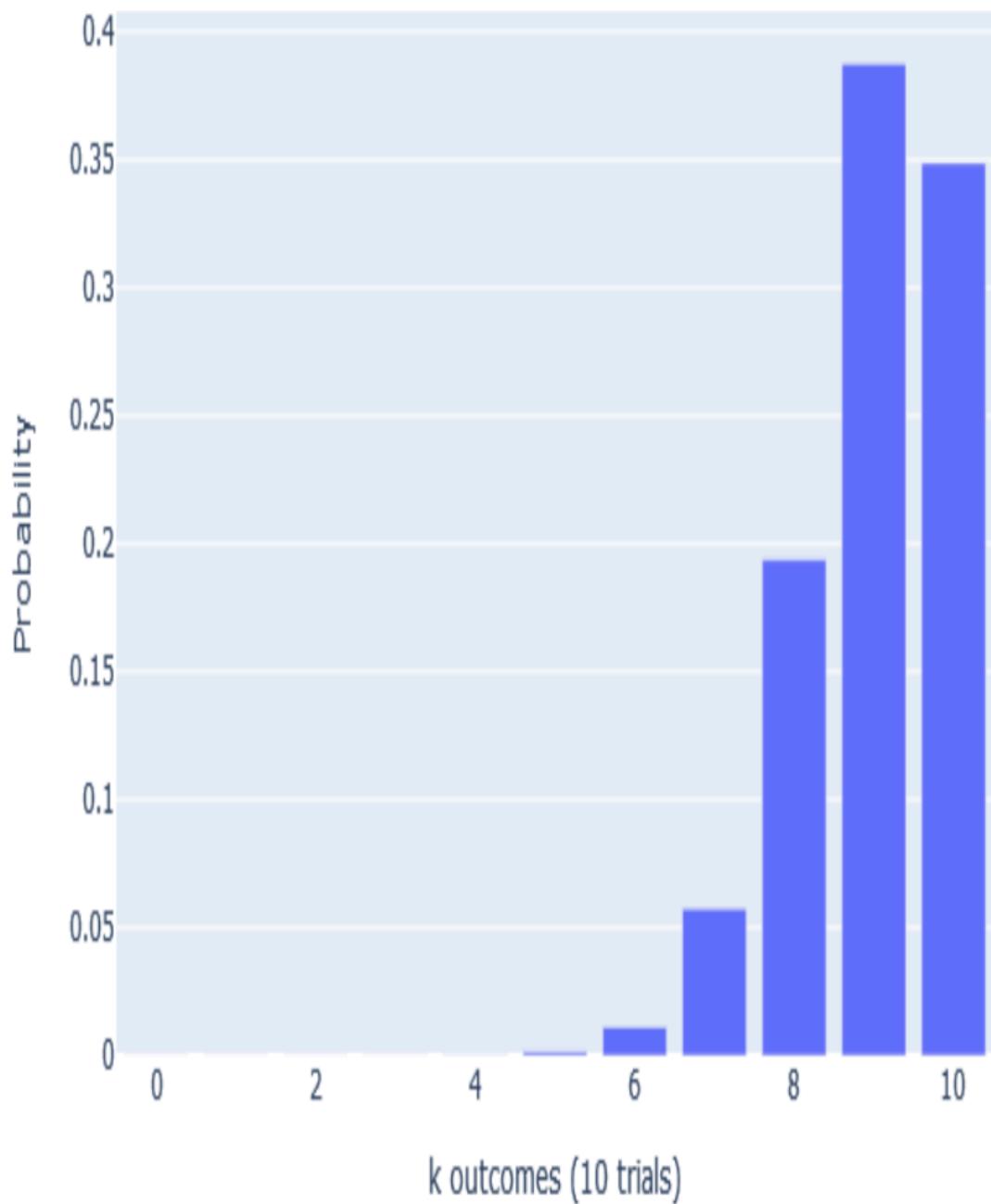


Figure 2-1. A binomial distribution

As you see above, we see the probability of k successes for each bar out of 10 trials. This binomial distribution assumes a probability p of 90%, meaning there is a .90 (or

90%) chance for a success to occur. If this is true, that means there is a .1937 probability we would get 8 successes out of 10 trials. The probability of getting 1 success out of 10 trials is extremely unlikely, .000000008999, hence why the bar is not even visible.

We can also calculate the probability of 8 or less successes by adding up bars for 8 or less successes. This would give us .2639 probability of 8 or less successes.

So how do we implement the binomial distribution? We can do it from scratch relatively easily (as shared in Appendix A, or we can use libraries like SciPy.

Example 2-2 shows how we use SciPy's `binom.pmf()` function (PMF stands for "probability mass function") to print all 11 probabilities for our binomial distribution from 0 to 10 successes.

Example 2-2. Using SciPy for the binomial distribution

```
from scipy.stats import binom

n = 10
p = 0.9

for k in range(n + 1):
    probability = binom.pmf(k, n, p)
    print("{} - {}".format(k, probability))

# OUTPUT:

# 0 - 9.99999999999996e-11
# 1 - 8.99999999999996e-09
# 2 - 3.64499999999996e-07
# 3 - 8.74800000000003e-06
# 4 - 0.0001377809999999999
# 5 - 0.0014880347999999988
# 6 - 0.01116026099999996
# 7 - 0.05739562800000001
# 8 - 0.1937102444999993
# 9 - 0.38742048900000037
# 10 - 0.3486784401000004
```

As you can see, we provide n as the number of trials, p as the probability of success for each trial, and k is the number of successes we want to look up the likelihood for. We iterate each number of successes x with the corresponding probability we would see that many successes. As we can see above in the output, the most likely number of successes is 9.

But if you add up the probability of 8 or less successes, we would get .2639. This means there is a 26.39% chance we would see 8 or less successes even if the

underlying success rate is 90%. So maybe the engineer is right. 26.39% chance is not nothing and certainly possible.

However we did make an assumption here in our model which we will discuss with the Beta Distribution.

BINOMIAL DISTRIBUTION FROM SCRATCH

Turn to Appendix A to learn how to build the Binomial Distribution from scratch rather than using Scikit-Learn.

Beta Distribution

What did I assume with my engine test model using the binomial distribution? Is there a parameter I assumed to be true and then built my entire model around it? Think carefully and read on.

What might be problematic about my binomial distribution above is I ASSUMED the underlying success rate is 90%. That's not to say my model is worthless. I just showed if the underlying success rate is 90%, there is a 26.39% chance I would see 8 or less successes with 10 trials. So the engineer is certainly not wrong that there could be an underlying success rate of 90%.

But let's flip the question and consider this: what if there are other underlying rates of success that would yield 8/10 successes besides 90%? Could we see 8/10 successes with an underlying 80% success rate? 70%? 30%? When we fix the 8/10 successes, can we explore the probabilities of probabilities?

Rather than create countless binomial distributions to answer this question, there is one tool that we can use. The **beta distribution** allows us to see the likelihood of different underlying probabilities for an event to occur given *alpha* successes and *beta* failures.

Here is a chart of the beta distribution given 8 successes and 2 failures in [Figure 2-2](#).

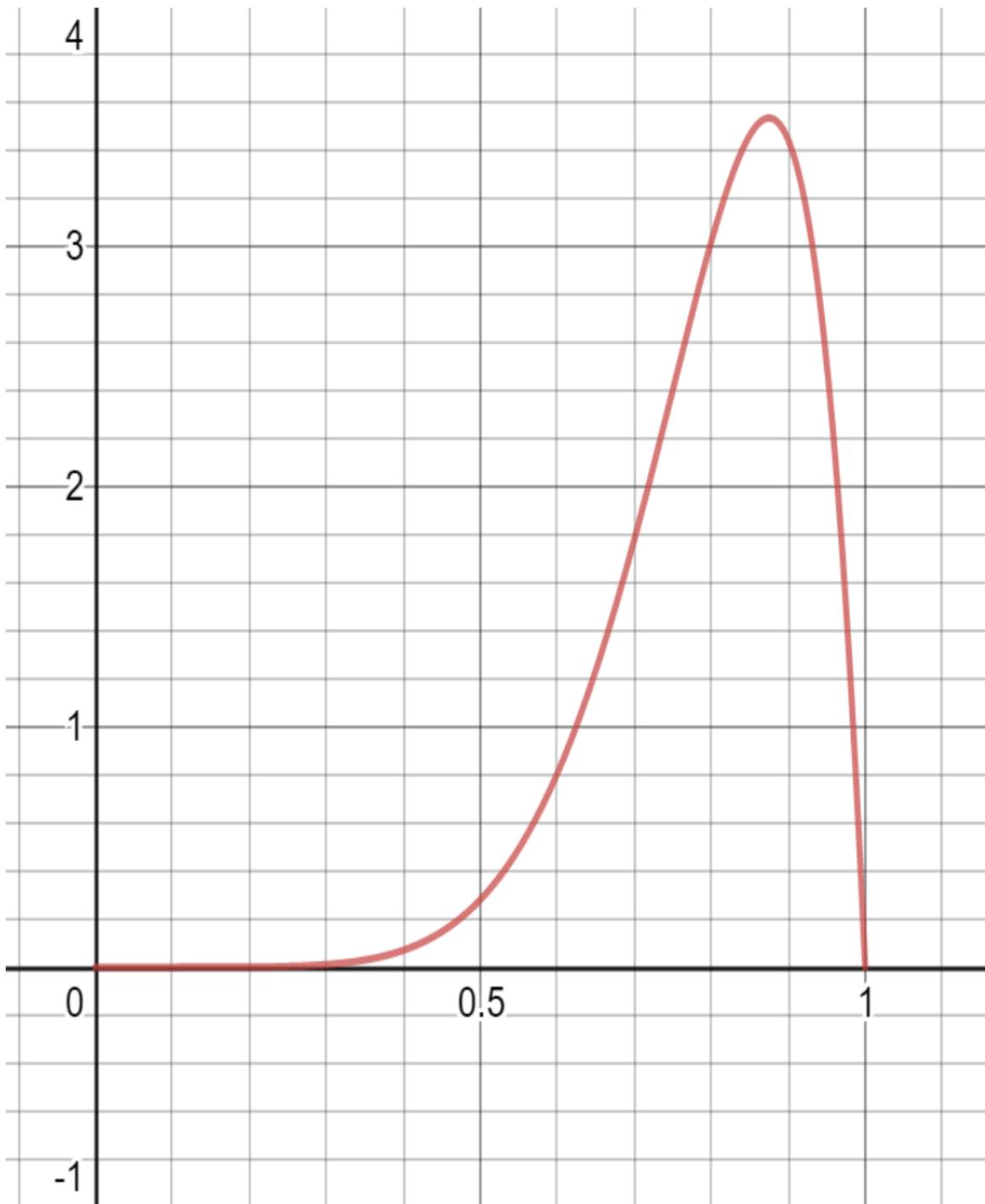


Figure 2-2. Beta distribution

BETA DISTRIBUTION ON DESMOS

If you want to interact with the Beta Distribution, a Desmos graph is provided here:
<https://www.desmos.com/calculator/xylhmcwo71>

Notice that the x-axis represents all underlying rates of success from 0.0 to 1.0 (0% to 100%), and the y-axis represents the likelihood of that probability given 8 successes and 2 failures. In other words, the beta distribution allows us to see the probabilities of probabilities given 8/10 successes. Think of it as a meta-probability so take your time grasping this idea!

Notice also that the beta distribution is a continuous function, meaning it forms a continuous curve of decimal values (as opposed to the tidy and discrete integers in the binomial distribution). This is going to make the math with the beta distribution a bit harder, as a given density value on the y-axis is not a probability. We instead find probabilities using areas under the curve.

The beta distribution is a type of **probability distribution**, which means the area under the entire curve is 1.0, or 100%. To find a probability, we need to find the area within a range. For example, if we want to evaluate the probability 8/10 successes would yield 90% or higher success rate, we need to find the area between 90% and 100%, which is 22.5%, as shaded in [Figure 2-3](#).

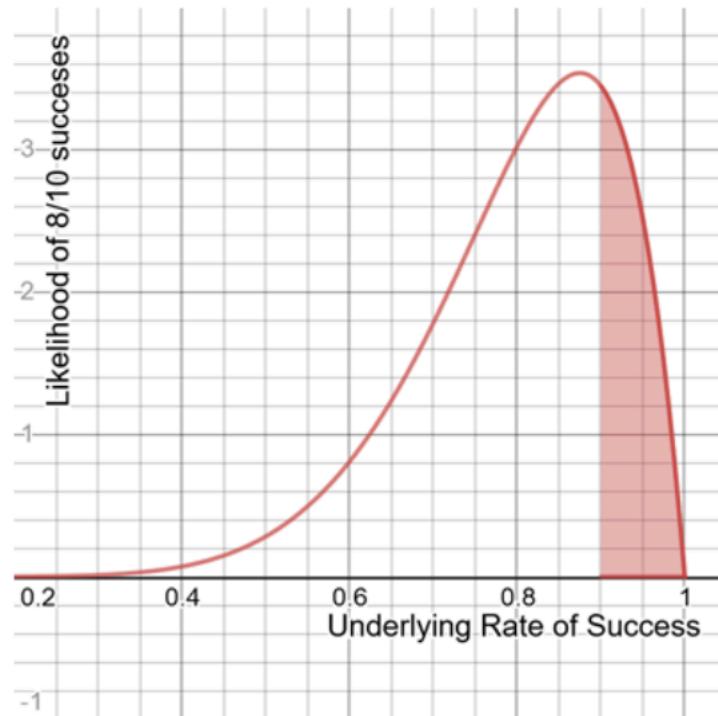


Figure 2-3. The area between 90% and 100%, which is 22.5%

As we did with the binomial distribution, we can use SciPy to implement the beta distribution. Every continuous probability distribution has a **cumulative density function (CDF)** which calculates the area up to a given x value. Let's say I wanted to calculate the area up to 90% (0.0 to .90) as shaded in [Figure 2-4](#).

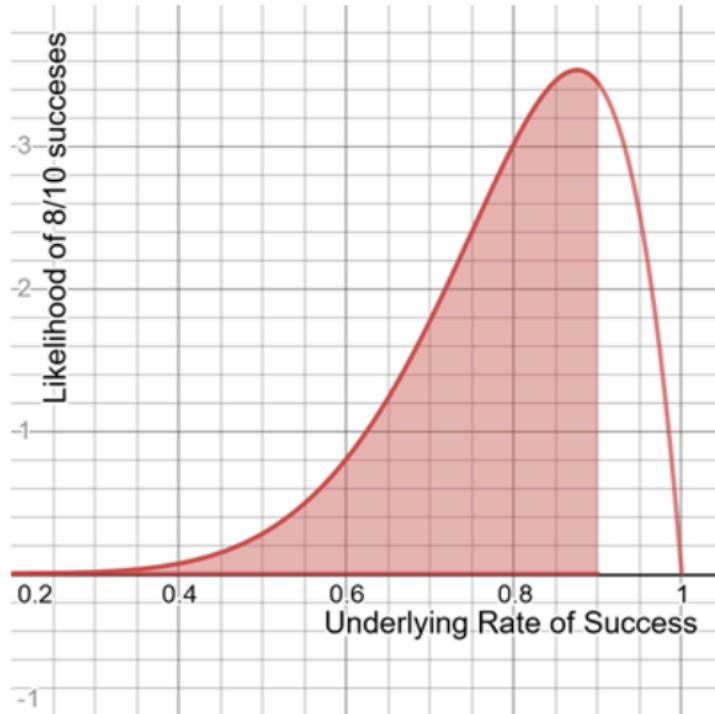


Figure 2-4. Calculating the area up to 90% (0.0 to .90)

It is easy enough to use SciPy with its `beta.cdf()` function, and the only parameters I need to provide are the x value, the number of successes a , and the number of failures b as shown in [Example 2-3](#).

Example 2-3. Beta distribution using SciPy

```
from scipy.stats import beta

a = 8
b = 2

p = beta.cdf(.90, a, b)

# 0.7748409780000001
print(p)
```

So according to our calculation above, there is a 77.48% chance the underlying probability of success is 90% or less.

How do we calculate the probability of success being 90% or more as shaded below in [Figure 2-5](#)?

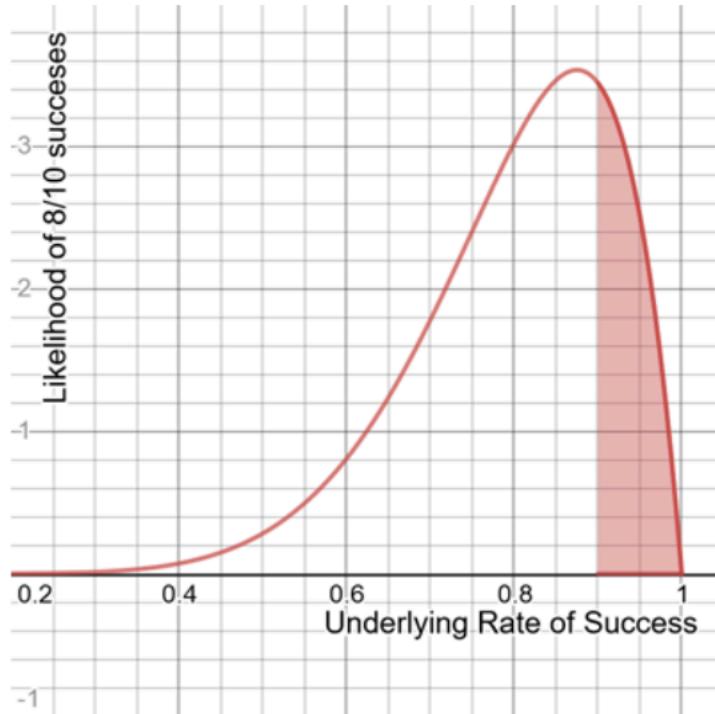


Figure 2-5. The probability of success being 90% or more

Our CDF only calculates area to the left of our boundary, not the right. Think about our rules of probability, and with a probability distribution the total area under the curve is 1.0. If we want to find the opposite probability of an event (greater than .90 as opposed to less than .90), just subtract the probability of being less than .90 from 1.0, and the remaining probability will capture being greater than .90. Here's a visual demonstrating how we do this subtraction in [Figure 2-6](#).

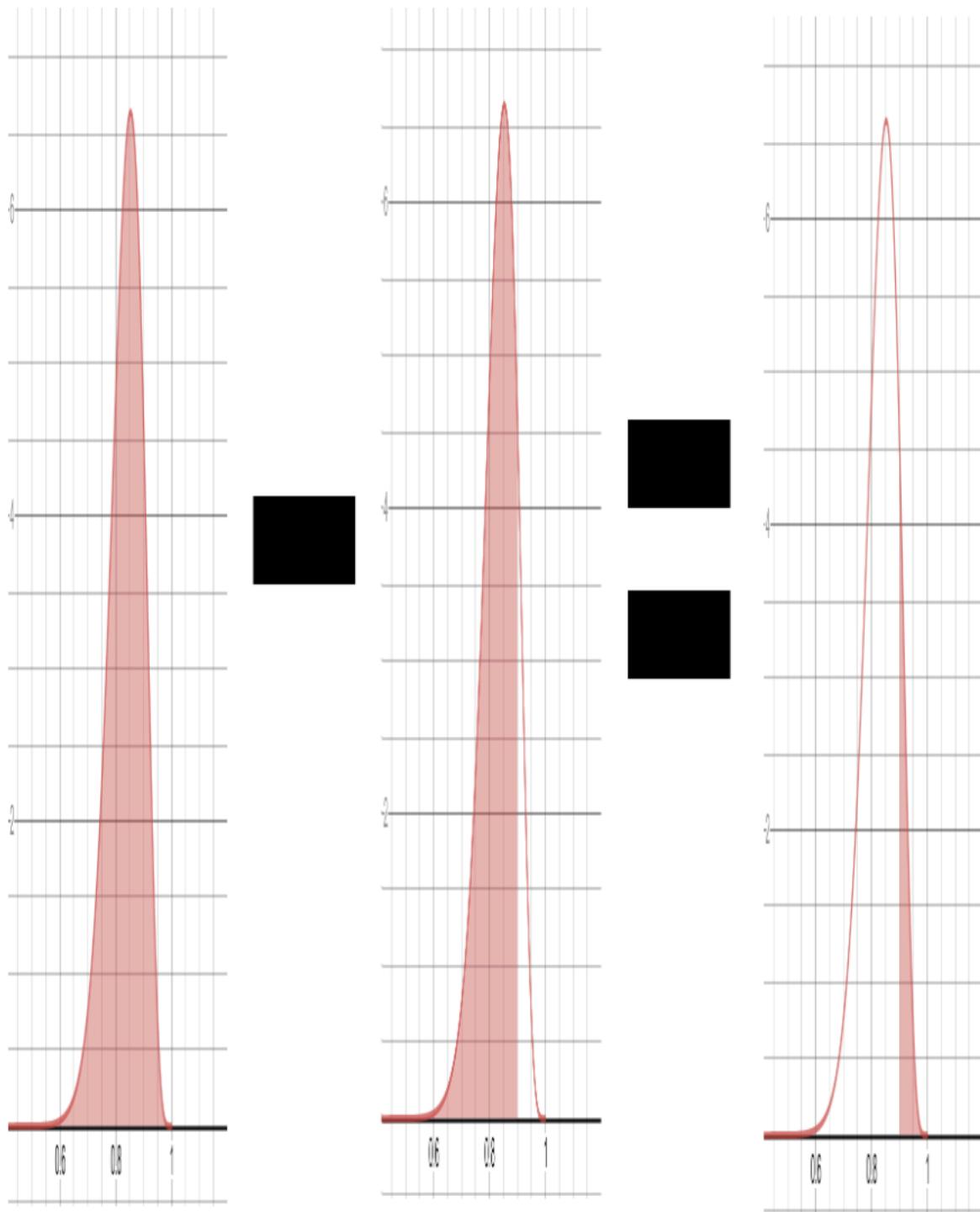


Figure 2-6. Finding the probability of success being greater than 90%

Example 2-4 shows how we calculate this subtraction operation in Python.

Example 2-4. Beta distribution using SciPy

```
from scipy.stats import beta
```

```
a = 8
b = 2

p = 1.0 - beta.cdf(.90, a, b)

# 0.22515902199999993
print(p)
```

So this means out of 8/10 successful engine tests, there is only a 22.5% chance the underlying success rate is 90% or greater. But there is about a 77.5% chance it is less than 90%. The odds are not in our favor here that our tests were successful, but we could gamble on that 22.5% chance with more tests if we are feeling lucky. If our CFO granted funding for 26 more tests resulting in 30 successes and 6 failures, here is what our Beta distribution would look now in [Figure 2-7](#).

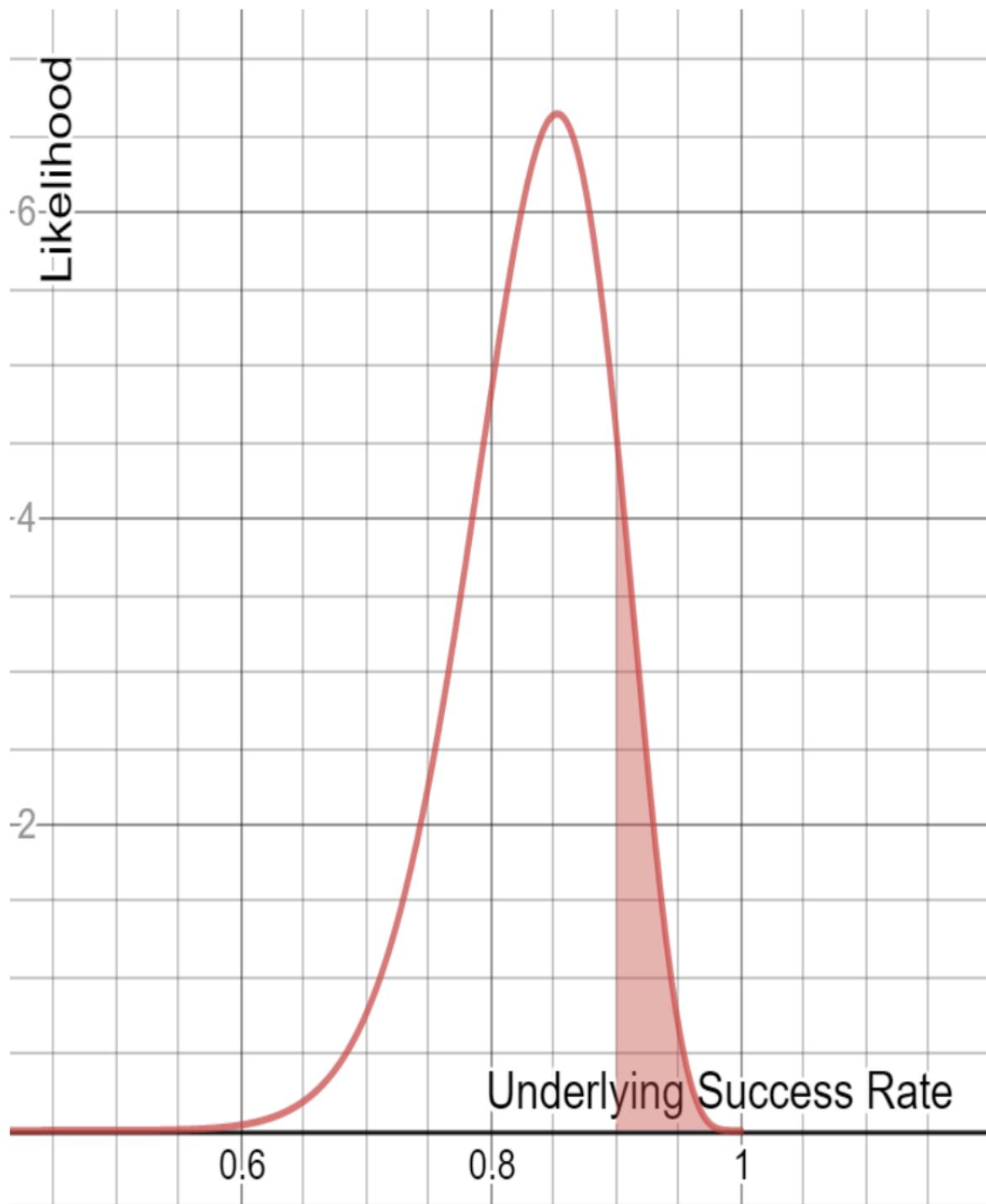


Figure 2-7. Beta distribution after 30 successes and 6 failures

Notice our distribution became narrower, thus becoming more confident that the underlying rate of success is in a smaller range. Unfortunately our probability of meeting our 90% success rate minimum has only decreased, going from 22.5% to 13.16% as shown in [Example 2-5](#).

Example 2-5. Beta distribution using SciPy

```
from scipy.stats import beta

a = 30
b = 6

p = 1.0 - beta.cdf(.90, a, b)

# 0.13163577484183708
print(p)
```

At this point, it might be a good idea to walk away and stop doing tests, unless you want to keep gambling against that 13.16% chance and hope the peak moves to the right.

Last but not least, how would we calculate an area in the middle? What if I want to find the probability my underlying rate of success is between 80% and 90% as shown in [Figure 2-8](#)?

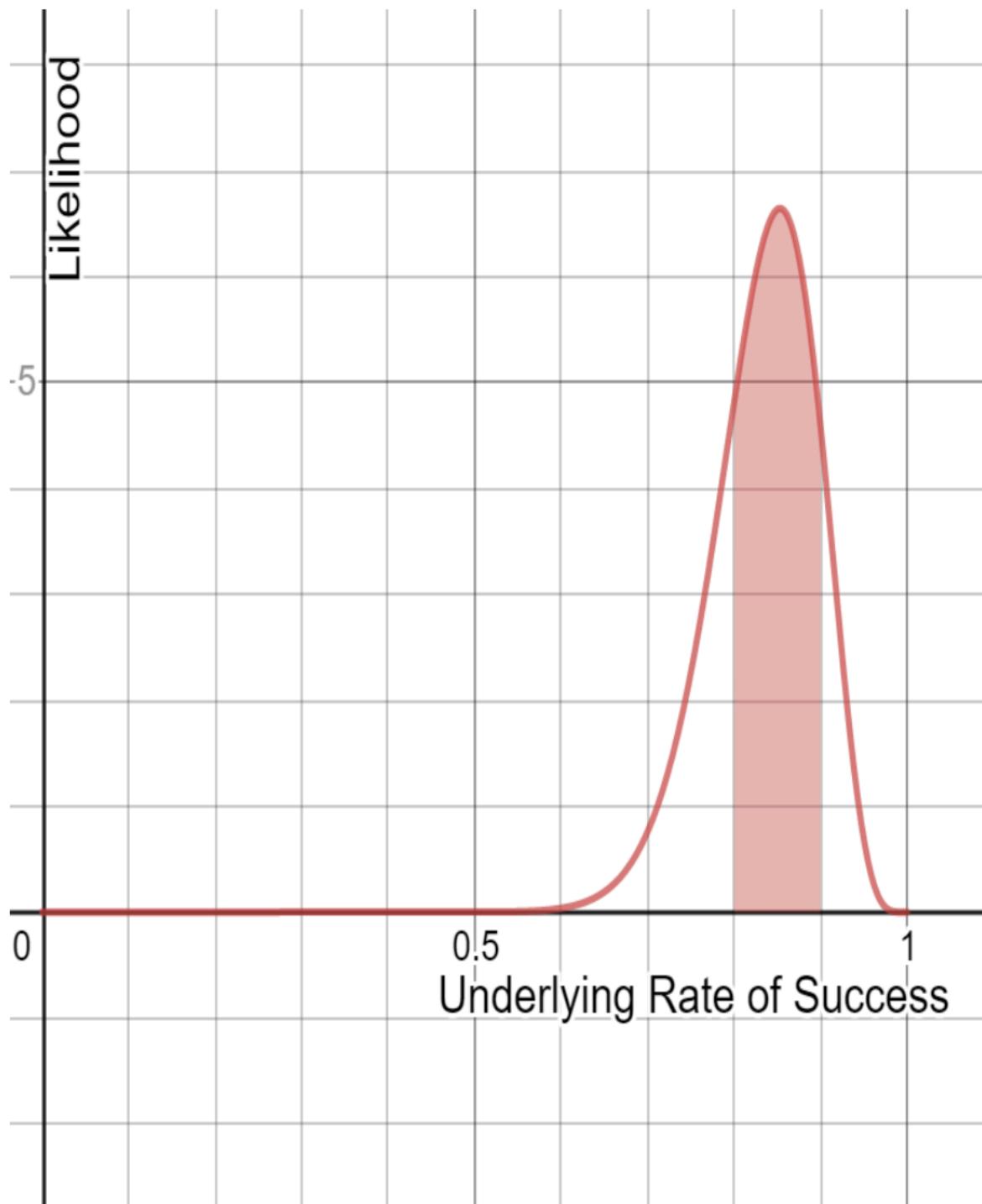


Figure 2-8. Probability the underlying rate of success is between 80% and 90%

Think carefully how you might approach this. What if we were to subtract the area behind .80 from the area behind .90 like in [Figure 2-9](#)?

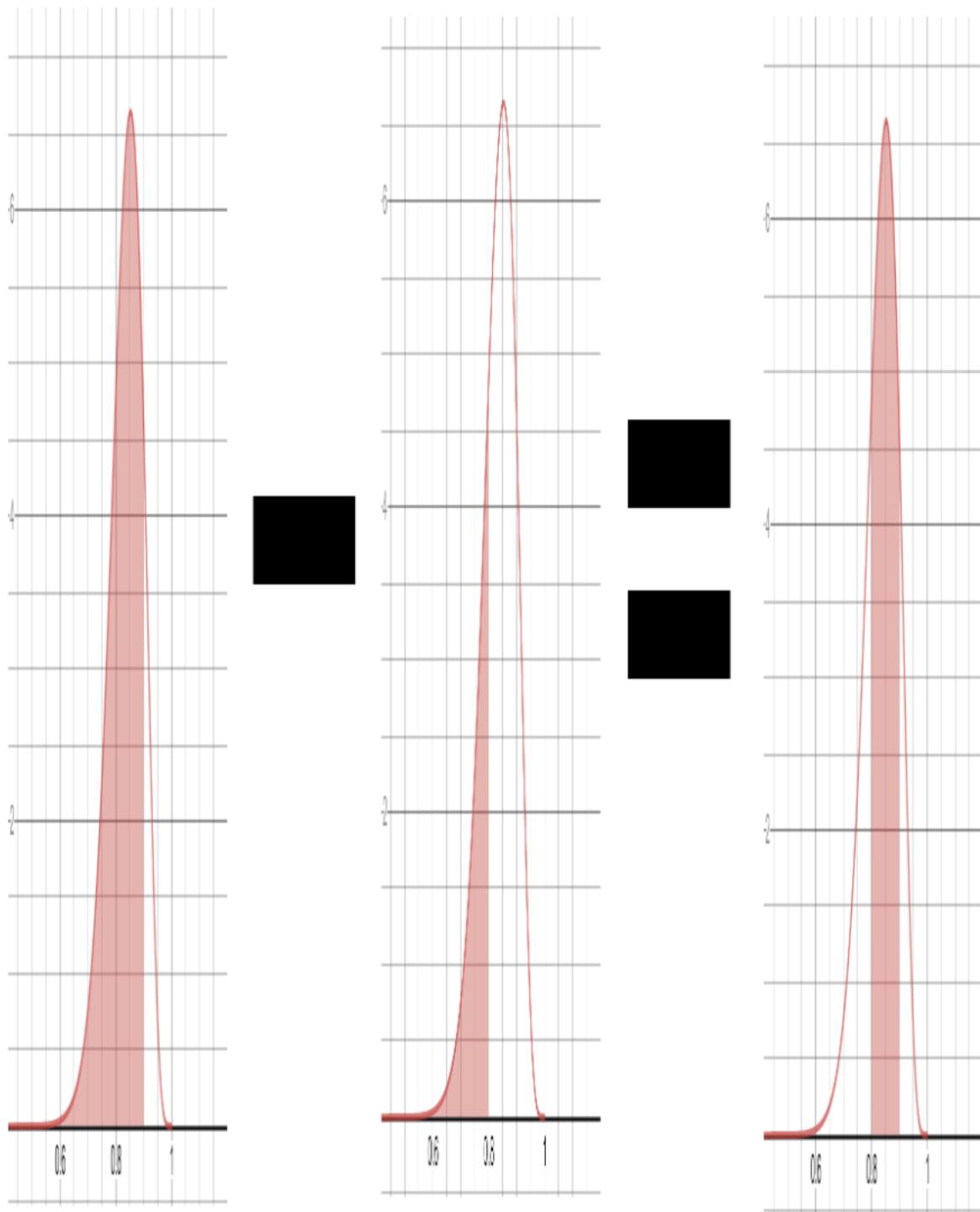


Figure 2-9. Obtaining the area between .80 and .90

Would that not give us the area between .80 and .90? Yes it would, and it would yield an area of .3386 or 33.86% probability. Here is how we would calculate it in Python in [Example 2-6](#).

Example 2-6. Beta distribution middle area using SciPy

```
from scipy.stats import beta

a = 8
b = 2

p = beta.cdf(.90, a, b) - beta.cdf(.80, a, b)

# 0.33863336200000016
print(p)
```

The beta distribution is a fascinating tool to measure the probability of an event occurring versus not occurring, based on a limited set of observations. It allows us to reason about probabilities of probabilities, and we can update it as we get new data. We can also use it for hypothesis testing, but we will put more emphasis on using the normal distribution and t-distribution for that purpose in [Chapter 3](#).

BETA DISTRIBUTION FROM SCRATCH

To learn how to implement the beta distribution from scratch, refer to Appendix A.

Conclusion

We covered a lot in this section! We not only covered the fundamentals of probability, its logical operators, and Bayes Theorem but we introduced probability distributions including the binomial and beta distributions. In the next chapter we will cover one of the more famous distributions, the normal distribution, and how it relates to hypothesis testing.

If you want to learn more about Bayesian probability and statistics, a great book is *Bayesian Statistics the Fun Way* by Will Kurt. There are also interactive Katacoda scenarios available on the O'Reilly platform. (NOTE TO EDITOR AND SELF: put links here?).

Exercises

1. There is a 30% chance of rain today, and a 40% chance your umbrella order will arrive on time. You are eager to walk in the rain today and cannot do so without either!

What is the probability it will rain AND your umbrella will arrive?

2. There is a 30% chance of rain today, and a 40% chance your umbrella order will arrive on time.

You will only be able to run errands if it does not rain or your umbrella arrives.

What is the probability it will not rain OR your umbrella arrives?

3. There is a 30% chance of rain today, and a 40% chance your umbrella order will arrive on time.

However, you found out if it rains there is only 20% chance your umbrella will arrive on time.

What is the probability it will rain and your umbrella will arrive on time?

4. You have 137 passengers booked on a flight from Las Vegas to Dallas.

However it is Las Vegas on a Sunday morning and you estimate each passenger is 40% likely to not show up.

You are trying to figure out how many seats to overbook so the plane does not fly empty.

How likely is it at least 50 passengers will not show up?

5. You flipped a coin 19 times and got heads 15 times and tails 4 times.

Do you think this coin has any good probability of being fair? Why or why not?

Chapter 3. Descriptive and Inferential Statistics

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at thomasnield@live.com.

Statistics is the practice of collecting and analyzing data to discover findings that are useful, or predict what causes those findings to happen. Probability often plays a large role in statistics, as we use data to estimate how likely an event is to happen.

It may not always get credit, but statistics is the heart of so many data-driven innovations. Machine learning in itself is a statistical tool, searching for possible hypotheses to correlate relationships between different variables in data.

However there are a lot of blind sides in statistics, even for professional statisticians. We can so easily get caught up in the data that we forget to ask what is not captured in the data. These concerns become all the more important as “big data”, “data mining”, and “machine learning” all accelerate the automation of statistical algorithms. Therefore it is important to have a solid foundation in statistics and hypothesis testing, so you do not treat these automations as black boxes as so many do.

In this section we will cover the fundamentals of statistics and hypothesis testing. Starting with descriptive statistics, we will learn common ways to summarize data. After that we will venture into inferential statistics where we try to uncover attributes of a population based on a sample.

What is Data?

It may seem odd I have to define data, something we all use and take for granted. But I think it needs to be done. Chances are if you asked any person what is data, they might answer to the effect of “you know... data! It’s... you know... information!” and not venture further than that. Nowadays it seems to be marketed as the be all and end all. The source of not just truth... but intelligence! It’s the fuel for “artificial intelligence” and it is believed that the more data you have, the more truth you have. Therefore you can never have enough data. It will unlock the secrets needed to redefine your business strategy and maybe even create artificial general intelligence. But let me offer a pragmatic perspective on what data is. Data is not important in itself. It’s the analysis of data (and how it is produced) that is the driver of all these innovations and solutions.

Imagine you were provided a photo of a family. Can you glean this family’s story based on this one photo? What if you had 20 photos? 200 photos? 2000 photos? How many photos do you need to know their story? Do you need photos of them in different situations? Alone and together? With relatives and friends? At home and at work?

Data is just like photographs; it provides snapshots of a story. The continuous reality and contexts are not fully captured, as well as the infinite number of variables that are driving that story. As we will discuss, data may be biased. It can have gaps and be missing relevant variables. Ideally we would love to have an infinite amount of data capturing an infinite number of variables, with so much detail we could virtually recreate reality and construct alternate ones! But is this possible? Currently, no. Not even the greatest supercomputers in the world combined can come close to capturing the entirety of the world as data.

Therefore we have to narrow our scope to make our objectives feasible. A few strategic photos of the father playing golf can easily tell us whether he is good at golf. But trying to decipher his entire life story just through photos? That might be impossible. There is so much that cannot be captured in snapshots. These practical concerns should also be applied when working with data projects, because data is actually just snapshots of a given time capturing only what it is aimed at (much like a camera). We need to keep our objectives focused, as this hones in on gathering data that is relevant and complete. If we make our objectives broad and open-ended, we can get into trouble with spurious findings and incomplete

datasets. This practice, known as **data mining**, has a time and place but it must be done carefully. We will revisit this at the end of the chapter.

Even with narrowly defined objectives, we can still run into problems with our data. Let's return to the question of determining whether a few strategic photos can tell whether the father is good at golf. Perhaps if you had a photo of him mid-swing you would be able to tell if he had good form. Or perhaps if you saw him cheering and fist-pumping at a hole, you can infer he got a good score. Maybe you can just take a photo of his scorecard! But it's important to note all these instances can be faked or taken out of context. Maybe he was cheering for someone else, or maybe the scorecard was not his or even forged. Just like these photographs, data does not capture context or truth. This is an incredibly important point because data provides clues, not truth. These clues can lead us to the truth or mislead us into erroneous conclusions.

This is why being curious about where data comes from is such an important skill. Ask questions on how the data was created, who it was created by, and what the data is not capturing. Nowadays it is too easy to get caught up in what the data says, and forgetting to ask where it came from. Even worse there are widespread sentiments that one can shovel data into machine learning algorithms and expect the computer to work it all out. But as the old adage goes, “garbage in, garbage out.” No wonder only 13% of machine learning projects succeed according to VentureBeat [1]. Successful machine learning projects put thought and analysis into the data, as well as what produced the data.

[1] <https://venturebeat.com/2019/07/19/why-do-87-of-data-science-projects-never-make-it-into-production/>

Descriptive versus Inferential Statistics

What comes to mind when you hear the word “statistics”? Is it calculating mean, median, mode, charts, bell curves, and other tools to describe data? This is the most commonly understood part of statistics called **descriptive statistics** and we use it to summarize data. After all, is it more meaningful to scroll through a million records of data or have it summarized? We will cover this area of statistics first.

Inferential statistics tries to uncover attributes about a larger population, often based on a sample. It is often misunderstood and less intuitive than descriptive

statistics. Oftentimes we are interested in studying a group that is too large to observe (e.g. average height of adolescents in North America) and we have to resort to using only a few members of that group to infer conclusions about them. As you can guess, this is not easy to get right. After all, we are trying to represent a population with a sample which may not be representative. We will explore these caveats along the way.

Populations, Samples, and Bias

Before we dive deeper into descriptive and inferential statistics, it might be a good idea to lay out some definitions and relate them with tangible examples.

A **population** is a particular group of interest we want to study, such as “all seniors over the age of 65 in the North America”, “all golden retrievers in Scotland,” or “current high school sophomores at Los Altos High School.” Notice how we have boundaries on defining our population. Some of these boundaries are broad and capture a large group over a vast geography or age group. Others are highly specific and small such as the sophomores at Los Altos High School. How you hone in on defining a population depends on what you are interested in studying.

A **sample** is a subset of the population that is ideally random and unbiased, which we use to infer attributes about the population. We often have to study samples because polling the entire population is not always possible. Of course, some populations are easier to get a hold of if they are small and accessible. But measuring all seniors over 65 in North America? That is unlikely to be practical!

POPULATIONS CAN BE ABSTRACT!

It is important to note that populations can be theoretical and not physically tangible. In these cases our “population” acts more like a sample from something abstract. Here’s my favorite example: we are interested in flights that depart between 2PM and 3PM at an airport, but we lack enough flights at that time to reliably predict how often these flights are late. Therefore we may treat this “population” as a sample instead... from an underlying population of all theoretical flights taking off between 2PM and 3PM.

Problems like this are why many researchers resort to simulations to generate data. Simulations can be useful but rarely are accurate, as simulations only capture so many variables and have assumptions built in.

If we are going to infer attributes about a population based on a sample, it's important the sample be as random as possible so we do not skew our conclusions. Here's an example. Let's say I'm a college student at Arizona State University. I want to find the average number of hours college students watch television per week in the United States. I walk right outside my dorm and start polling random students walking by, finishing my data gathering in a few hours. What's the problem here?

The problem is our student sample is going to have **bias**, meaning it skews our findings by overrepresenting a certain group at the expense of other groups. My study defined the population to be "college students in the United States," not "college students at Arizona State University." I am only polling students at one specific university to represent all college students in the entire United States! Is that really fair?

It is unlikely all colleges across the country homogeneously have the same student attributes. What if Arizona State students watch far more TV than other students at other universities? Would that not distort the results using them to represent the entire country? Maybe this is possible because it is usually too hot to go outside in Tempe, Arizona. Therefore TV is a common pastime (anecdotally I would know, I lived in Phoenix for many years). Other college students in milder climates may do more outdoor activities and watch less TV.

This is just one possible variable showing why it's a bad idea to represent college students across the entire United States with just a sample of students from one university. Ideally, I should be randomly polling college students all over the country at different universities. That way I have a more representative sample.

However, bias is not always geographic. Let's say I put a sincere effort to poll students across the United States. I arrange a social media campaign to have a poll shared by various universities on Twitter and Facebook. This way their students see it and hopefully fill it out. I get hundreds of responses on students' TV habits across the country and feel I've conquered the bias beast... or have I?

What if students who are on social media enough to see the poll are also likely to watch more TV? If they are on social media a lot, they probably do not mind recreational screen time. It's easy to imagine they have Netflix and Hulu ready to stream on that other tab! This particular type of bias where a specific group is more likely to include themselves in a sample is known as **self-selection bias**.

SAMPLES AND BIAS IN MACHINE LEARNING

These problems with sampling and bias extends to machine learning as well. Whether it is linear regression, logistic regression, or neural networks, a sample of data is used to infer predictions. If that data is biased then it will steer the machine learning algorithm to make biased conclusions.

There are many documented cases of this. Criminal justice has been a precarious application of machine learning because it has repeatedly shown to be biased in every sense of the word, discriminating against minorities due to minority-heavy datasets. In 2017, Volvo tested “self-driving cars” that were trained on datasets capturing deer, elk, and caribou. However it had no driving data in Australia and therefore could not recognize kangaroos, much less make sense of their jumping movements! Both of these are examples of biased data.

Darn it! You just can't win, can you?! If you think about it long enough, data bias just feels inevitable! And oftentimes it is. So many **confounding variables**, or factors we did not account for, can influence our study. This problem of data bias is expensive and difficult to overcome, and machine learning is especially vulnerable to it.

The way to overcome this problem is to truly at random select students from the entire population, and they cannot elect themselves into or out of the sample voluntarily. This is the most effective way to mitigate bias, but as you can imagine it takes a lot of coordinated resources to do.

A WHIRLWIND TOUR OF BIAS TYPES

As humans, we are strangely wired to be biased. We look for patterns even if they do not exist. Perhaps this was a necessity for survival in our early history, as finding patterns make hunting, gathering, and farming more productive.

There are many types of bias, but they all have the same effect of distorting findings. **Confirmation bias** is only gathering data that supports your belief, which can even be done unknowingly. An example of this is only following social media accounts you politically agree with, reinforcing your beliefs rather than challenging them.

We just discussed **self-selection bias**, which is when certain types of subjects are more likely to include themselves in the experiment. Walking onto a flight and polling the customers if they like the airline over other airlines, and using that to rank customer satisfaction amongst all airlines, is silly. Why? Many of those customers are likely repeat customers, and they have created self-selection bias. We do not know how many are first-time fliers or repeat fliers, and of course the latter is going to prefer that airline over other airlines. Even first-time fliers may be biased and self-selected, because they chose to fly that airline and we are not sampling all airlines.

Survival bias only captures living and survived subjects, while the deceased ones are never accounted for. It is the most fascinating type of bias in my opinion, because the examples are so diverse and unobvious.

Probably the most famous example of survival bias is this: The Royal Air Force in WWII was having trouble with German enemy fire causing casualties on its bombers. Their initial solution was to armor bombers where bullet holes were found, reasoning this would improve their likelihood of survival. However a mathematician named Abraham Wald pointed to them this was fatally wrong. He proposed armoring areas on the aircraft that *did not* have bullet holes. Was he insane? Far from. The bombers that returned obviously did not have fatal damage where bullet holes were found. How do we know this? Because they returned from their mission! But what of the aircraft that did not return? Where were they hit? Wald's theory was likely in the areas untouched in the aircraft *that survived* and returned to base, and this proved right. Those areas without bullet holes were armored and the survivability of

aircraft and pilots increased. This unorthodox observation is credited with turning the tide of war in favor of the Allies.

There are other fascinating examples of survivor bias that are less obvious. Many management consulting companies and book publishers like to identify traits of successful companies/individuals and use them as predictors for future successes. These works are pure survival bias (and [XKCD has a funny cartoon about this](#)). These works do not account for companies/individuals that failed in obscurity, and these “success” qualities may be commonplace with failed ones as well. We just have not heard about them because they never had a spotlight.

An anecdotal example that comes to my mind is Steve Jobs. On many accounts, he was said to be passionate and hot-tempered... but he also created one of the most valuable companies of all time. Therefore some people believed being passionate and even temperamental might correlate with success. Again, this is survivor bias. I am willing to bet there are many companies run by passionate leaders that failed in obscurity, but we fixate and latch onto outlier success stories like Apple.

Lastly, in 1987 there was a veterinary study showing that cats who fell from 6 stories or less had greater injuries than those that fell more than 6 stories. Prevailing scientific theories postulated that cats righted themselves at about 5 stories, giving them enough time to brace for impact and inflict less injury. But then a publication *The Straight Dope* posed an important question: what happened to the dead cats? People are not likely to bring a dead cat to the vet and therefore it is unreported how many cats died from higher falls. In the words of Homer Simpson, "D'oh!".

Alright, enough talk about populations, samples, and bias. Let's move onto some math and descriptive statistics. Just remember that math and computers do not recognize bias in your data. That is on you to detect as a good data science professional! Always ask questions about how the data was obtained, and then scrutinize how that process could have biased the data.

Descriptive Statistics

Descriptive statistics is the area most people are familiar with. We will touch on the basics like mean, median, and mode followed by variance, standard deviation, and the normal distribution.

Mean and Weighted Mean

The **mean** is the average of a set of values. The operation is simple to do: sum the values and divide by the number of values. The mean is useful because it shows where the “center of gravity” exists for an observed set of values.

The mean is calculated the same way for both populations and samples.

Example 3-1 shows a sample of 8 values and how to calculate their mean in Python.

Example 3-1. Calculating mean in Python

```
# Number of pets each person owns
sample = [1, 3, 2, 5, 7, 0, 2, 3]

mean = sum(sample) / len(sample)

print(mean) # prints 2.875
```

As you can see, we polled 8 people and the number of pets they own. The sum of the sample is 23 and the number of items in the sample is 8, so this gives us a mean of 2.875 as $23/8 = 2.875$.

There are two versions of the mean you will see: the sample mean \bar{x} and the population mean μ as expressed below:

$$\bar{x} = \frac{x_1 + x_2 + x_3 + \dots + x_n}{n} = \sum \frac{x_i}{n}$$

$$\mu = \frac{x_1 + x_2 + x_3 + \dots + x_n}{N} = \sum \frac{x_i}{N}$$

Recall the summation symbol \sum means add all the items together. The n and the N represent the sample and population size respectively, but mathematically they represent the same thing: the number of items. The same goes for calling the sample mean \bar{x} (“x-bar”) and the population mean μ (“mu”). Both \bar{x} and μ are the same calculation, just different names depending on whether it’s a sample or population we are working with.

The mean is likely familiar to you, but here's something less known about the mean: the mean is actually a weighted average called the **weighted mean**. The mean we commonly use gives equal importance to each value. But we can manipulate the mean and give each item a different weight.

$$\text{weighted mean} = \frac{(x_1 \cdot w_1) + (x_2 \cdot w_2) + (x_3 \cdot w_3) + \dots (x_n \cdot w_n)}{w_1 + w_2 + w_3 + \dots + w_n}$$

This can be helpful when we want some values to contribute to the mean more than others. A common example of this is weighting academic exams to give a final grade. If you have 3 exams and a final exam, and we give each of the 3 exams 20% weight and the final exam 40% weight of the final grade, here is how we express it in [Example 3-2](#).

Example 3-2. Calculating a weighted mean in Python

```
# Three exams of .20 weight each and final exam of .40 weight
sample = [90, 80, 63, 87]
weights = [.20, .20, .20, .40]

weighted_mean = sum(s * w for s,w in zip(sample, weights)) / sum(weights)

print(weighted_mean) # prints 81.4
```

We weight each exam score through multiplication accordingly and instead of dividing by the value count, we divide by the sum of weights. Weightings don't have to be percentages, as any numbers used for weights will end up being proportionalized. In [Example 3-3](#), we weight each exam with "1" but weight the final exam with "2", giving it twice the weight of the exams. We will still get the same answer of 81.4 as these values will still be proportionalized.

Example 3-3. Calculating a weighted mean in Python

```
# Three exams of .20 weight each and final exam of .40 weight
sample = [90, 80, 63, 87]
weights = [1.0, 1.0, 1.0, 2.0]

weighted_mean = sum(s * w for s,w in zip(sample, weights)) / sum(weights)

print(weighted_mean) # prints 81.4
```

Median

The **median** is the middle-most value in a set of ordered values. You sequentially order the values, and the median will be the center-most value. If you have an even number of values, you average the two center-most values. We can see in **Example 3-4** the median number of pets owned in our sample is 7.

```
0, 1, 5, *7*, 9, 10, 14
```

Example 3-4. Calculating the median in Python

```
# Number of pets each person owns
sample = [0, 1, 5, 7, 9, 10, 14]

def median(values):
    ordered = sorted(values)
    print(ordered)
    n = len(ordered)
    mid = int(n / 2) - 1 if n % 2 == 0 else int(n/2)

    if n % 2 == 0:
        return (ordered[mid] + ordered[mid+1]) / 2.0
    else:
        return ordered[mid]

print(median(sample)) # prints 5
```

The median can be a helpful alternative to the mean when data is skewed by **outliers**, or values that are extremely large and small compared to the rest of the values. Here's an interesting anecdote to understand why. In 1986, the mean annual starting salary of geography graduates in the University of North Carolina was \$250,000. Other universities averaged \$22,000. Wow, UNC must have an amazing geography program!

But in reality, what was so lucrative about UNC's geography program? Well... they had Michael Jordan as one of their graduates. One of the most famous NBA players of all time indeed graduated with a geography degree from University of North Carolina. However he started his career playing basketball, not studying maps. Obviously this is a confounding variable which has created a huge outlier, and it majorly skewed the income average.

This is why the median can be preferable in outlier-heavy situations (such as income-related data) over the mean. It is less sensitive to outliers and cuts data strictly down the middle based on their relative order, rather than where they fall

exactly on a number line. When your median is very different from mean, that means you have a skewed dataset with outliers.

THE MEDIAN IS A QUANTILE

There is a concept of **quantiles** in descriptive statistics. The concept of quantiles is essentially the same as a median, just cutting the data in other places besides the middle. The median is actually the 50% quantile, or the value where 50% of ordered values are behind it. Then there is the 25%, 50%, 75% quantiles which are known as **quartiles** as they cut data in 25% increments.

Mode

The **mode** is the most frequently occurring set of values. It primarily becomes useful when your data is repetitive and you want to find which values occur the most frequently.

When no value occurs more than once, there is no mode. When two values occur with an equal amount of frequency, then the dataset is considered **bimodal**. In [Example 3-5](#) we calculate the mode for our pet dataset, and sure enough we see this is bimodal as both 2 and 3 occur the most (and equally) as often.

Example 3-5. Calculating the mode in Python

```
# Number of pets each person owns
from collections import defaultdict

sample = [1, 3, 2, 5, 7, 0, 2, 3]

def mode(values):
    counts = defaultdict(lambda: 0)

    for s in values:
        counts[s] += 1

    max_count = max(counts.values())
    modes = [v for v in set(values) if counts[v] == max_count]
    return modes

print(mode(sample)) # [2, 3]
```

In practicality, the mode is often not used a lot unless your data is repetitive. This is commonly encountered with integers, categories, and other discrete variables.

Variance and Standard Deviation

When we start talking about variance and standard deviation, this is where it gets interesting. One thing that confuses people with variance and standard deviation is there are some calculation differences for the sample versus the population. We will do our best to cover these differences clearly.

Population Variance and Standard Deviation

In describing data, we are often interested in measuring the differences between the mean and every data point. This gives us a sense of how “spread out” the data is.

Let’s say I’m interested in studying the number of pets owned by members of my work staff (note that I’m defining this as my population, not a sample). I have 7 people on my staff.

I take the mean of all the numbers of pets they own, and I get 6.571. Let’s subtract this mean from each value. This will show us how far each value is from the mean as shown in [Table 3-1](#).

Table 3-1. Number of pets my staff owns

Value	Mean	Difference
0	6.571	-6.571
1	6.571	-5.571
5	6.571	-1.571
7	6.571	0.429
9	6.571	2.429
10	6.571	3.429
14	6.571	7.429

Let’s visualize this on a number line with “X” showing the mean in [Figure 3-1](#).

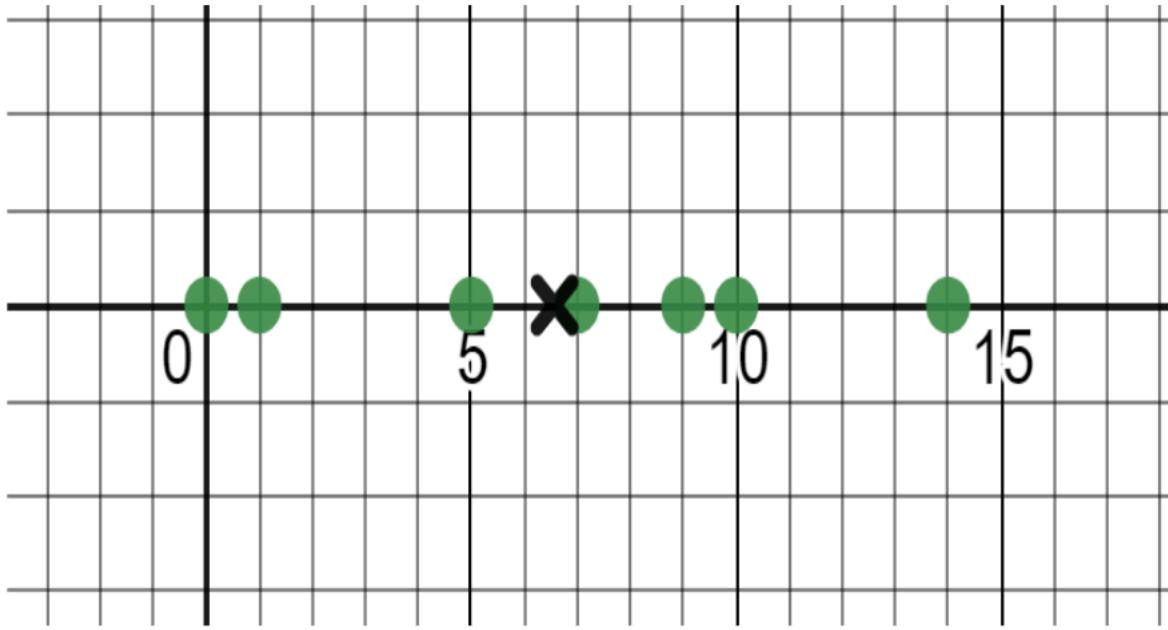


Figure 3-1. Visualizing the spread of our data, where “X” is the mean

Hmmm... now consider why the information above can be useful. The differences give us a sense of how “spread out” the data is, and how far values are from the mean. Is there a way we can consolidate these differences into a single number to quickly describe how spread out the data is?

You may be tempted to take the average of the differences, but the negatives and positives will cancel each other out when they are summed. We could sum the absolute values (rid the negative signs and make all values positive). An even better approach would be to square these differences before summing them. This not only rids the negative values (because squaring a negative number makes it positive), but it amplifies larger differences. After that, average the squared differences. This will give us the **variance**, a measure of how spread out our data is.

Here is a math formula showing how to calculate variance:

$$\text{population variance} = \frac{(x_1 - \text{mean})^2 + (x_2 - \text{mean})^2 + \dots + (x_n - \text{mean})^2}{N}$$

More formally, here is the variance for a population.

$$\sigma^2 = \frac{\sum (x_i - \mu)^2}{N}$$

Calculating the population variance of our pet example in Python is shown in [Example 3-6](#).

Example 3-6. Calculating variance in Python

```
# Number of pets each person owns
data = [0, 1, 5, 7, 9, 10, 14]

def variance(values):
    mean = sum(values) / len(values)
    _variance = sum((v - mean) ** 2 for v in values) / len(values)
    return _variance

print(variance(data)) # prints 21.387755102040813
```

So the variance for number of pets owned by my office staff is 21.387755. Okay, but what does it exactly mean? It's reasonable to conclude that a higher variance means more spread, but how do we relate this back to our data? This number is larger than any of our observations because we did a lot squaring and summing, putting it on an entirely different metric. So how do we squeeze it back down so it's back on the scale we started with?

The opposite of a square is a square root, so let's take the square root of the variance which gives us the **standard deviation**. This is the variance scaled into a number expressed in terms of "number of pets" which makes it a bit more meaningful.

$$\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}$$

To implement in Python, we can reuse the `variance()` function and `sqrt()` its result. We now have a `std_dev()` function shown in [Example 3-7](#).

Example 3-7. Calculating standard deviation in Python

```
from math import sqrt

# Number of pets each person owns
data = [0, 1, 5, 7, 9, 10, 14]

def variance(values):
    mean = sum(values) / len(values)
    _variance = sum((v - mean) ** 2 for v in values) / len(values)
    return _variance

def std_dev(values):
```

```

    return sqrt(variance(values))

print(std_dev(data)) # prints 4.624689730353898

```

Running the code in [Example 3-7](#), you will see our standard deviation is approximately 4.62 pets. So we can express our spread on a scale we started with, and this makes our variance a bit easier to interpret. We will see some important applications of the standard deviation in this chapter as well as future chapters, including [Chapter 5](#).

WHY THE SQUARE?

If the exponent in σ^2 bothers you, it's because it is prompting you to take the square root of it to get the standard deviation. It's like a little reminder you are dealing with squared values that need to be square-rooted.

Sample Variance and Standard Deviation

In the previous section we talked about variance and standard deviation for a population. However there is an important tweak we need to apply to these two formulas when we calculate for a sample:

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

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

Did you catch the difference? When we average the squared differences, we divide by $n - 1$ rather than the total number of items n . Why would we do this? We do this to decrease any bias in a sample, and not underestimate the variance of the population based on our sample. By counting values short of 1 item in our divisor, we increase the variance and therefore capture greater uncertainty in our sample.

WHY DO WE SUBTRACT 1 FOR SAMPLE SIZE?

On YouTube, Josh Starmer has an excellent video series called StatQuest. In one, he gives an excellent breakdown why we treat samples differently in calculating variance and subtract one element from the number of items.

<https://www.youtube.com/watch?v=sHRBg6BhKjI>

If our pets data were a sample, not a population, we should make that adjustment accordingly. In [Example 3-8](#), I modify my previous `variance()` and `std_dev()` Python code to optionally provide a parameter `is_sample`, which if `True` will subtract 1 from the divisor in the variance.

Example 3-8. Calculating standard deviation in Python

```
from math import sqrt

# Number of pets each person owns
data = [0, 1, 5, 7, 9, 10, 14]

def variance(values, is_sample: bool = False):
    mean = sum(values) / len(values)
    _variance = sum((v - mean) ** 2 for v in values) /
        (len(values) - (1 if is_sample else 0))

    return _variance

def std_dev(values, is_sample: bool = False):
    return sqrt(variance(values, is_sample))

print("VARIANCE = {}".format(variance(data, is_sample=True))) # 24.95238095238095
print("STD DEV = {}".format(std_dev(data, is_sample=True))) # 4.99523582550223
```

Notice in [Example 3-8](#) my variance and standard deviation have increased compared to previous examples that treated them as a population, not a sample. Recall in [Example 3-7](#) that the standard deviation was about 4.62 treating as a population. But here treating as a sample (by subtracting 1 from the variance denominator) we get approximately 4.99. This is correct as a sample could be biased and imperfect representing the population. Therefore we increase the variance (and thus the standard deviation) to increase our estimate how spread out the values are. A larger variance/standard deviation shows less confidence with a larger range.

To recap, just like the mean (\bar{x} for sample and μ for population), you will often see certain symbols for variance and standard deviation. The standard deviation for a sample and mean are specified by s and sigma σ respectively. Here again are the sample and population standard deviation formulas:

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

$$\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}$$

The variance will be the square of these two formulas, undoing the square root. Therefore the variance for sample and population are s^2 and σ^2 respectively.

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

$$\sigma^2 = \frac{\sum (x_i - \mu)^2}{N}$$

The square helps imply that a square root should be taken to get the standard deviation.

The Normal Distribution

We touched on probability distributions in the last chapter, particularly the binomial distribution and beta distribution. However the most famous distribution of all is the normal distribution. The **normal distribution**, also known as the **Gaussian distribution**, is a symmetrical bell-shaped distribution that has most mass around the mean, and its spread is defined as a standard deviation. The “tails” on either side become thinner as you move away from the mean.

Here is a normal distribution for golden retriever weights. Notice how most of the mass is around the mean of 64.43 pounds. Let’s explore how we encounter the normal distribution in [Figure 3-2](#).

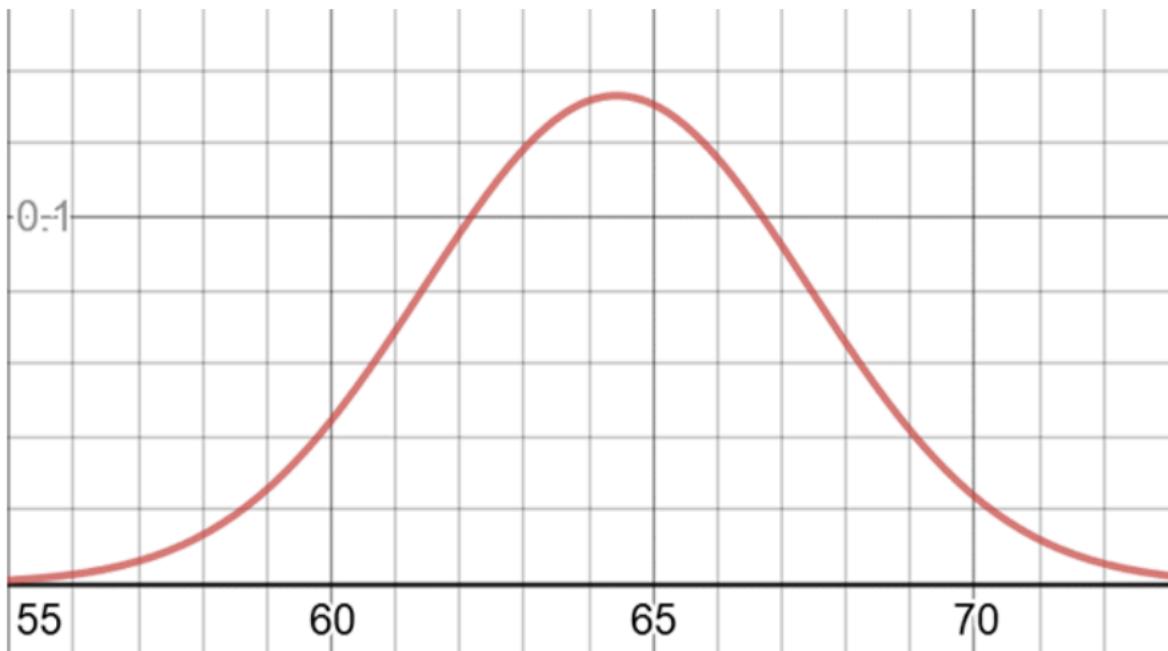


Figure 3-2. A normal distribution

Discovering the Normal Distribution

The normal distribution is seen a lot in nature, engineering, science, and other domains. How do we discover it? Let's say we sample the weight of 50 adult golden retrievers and plot them on a number line as shown in [Figure 3-3](#).

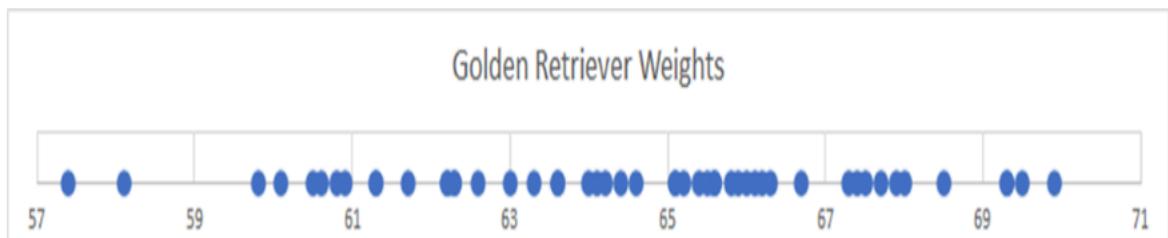


Figure 3-3. A sample of 50 golden retriever weights

Notice how we have more values towards a center, but as we move further left or right we see less values. Based on our sample, it seems highly unlikely we would see a golden retriever with a weight of 57 or 71. But having a weight of 64 or 65? Yeah that certainly seems likely.

Is there a better way to visualize this likelihood to see which golden retriever weights we are more likely to see sampled from the population? We can try to create a **histogram**, which buckets (or “bins”) up values based on numeric ranges of equal length, and then uses a bar chart showing the number of values within

each range. In <<vtFcGtuatD>> we create a histogram that bins up values in ranges of .5 pounds.

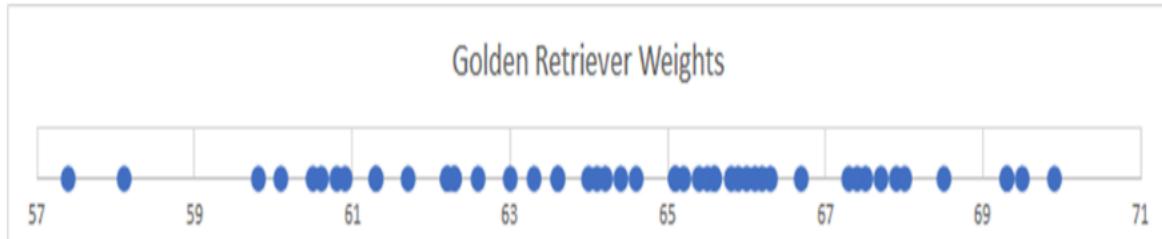


Figure 3-4. A histogram of golden retriever weights

This histogram above does reveal any meaningful shape to our data. The reason is because our bins are too small. We do not have an extremely large or infinite amount of data to meaningfully have enough points in each bin. Therefore we will have to make our bins larger. Let's make the bins each have a length of 3 pounds in Figure 3-5.

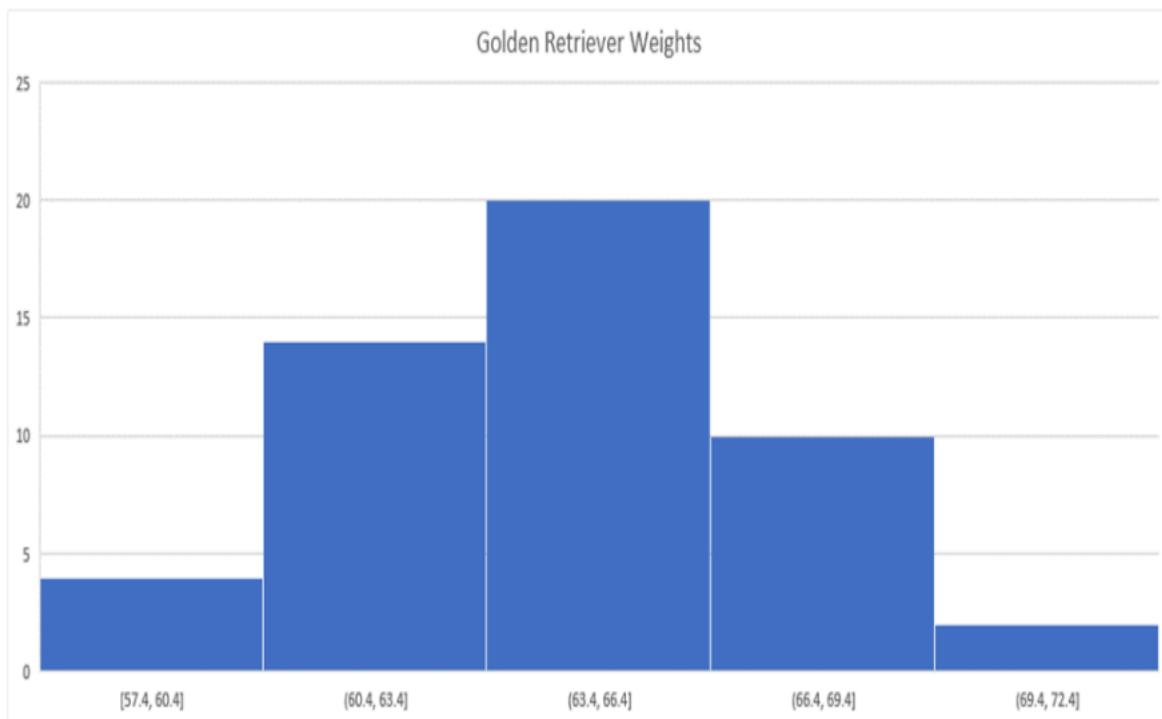


Figure 3-5. A more productive histogram

Alright now we are getting somewhere! As you can see above, if we get the bin sizes just right (in this case each has a range of 3 lbs), we start to get a meaningful bell shape to our data. It's not a perfect bell shape because our samples are never going to be perfectly representative of the population, but this is likely evidence

our sample follows a normal distribution. If we fit a histogram with adequate bin sizes, scale it so it has an area of 1.0 (which a probability distribution requires), we roughly see a bell curve representing our sample. Let's show it alongside our original data points in [Figure 3-6](#).

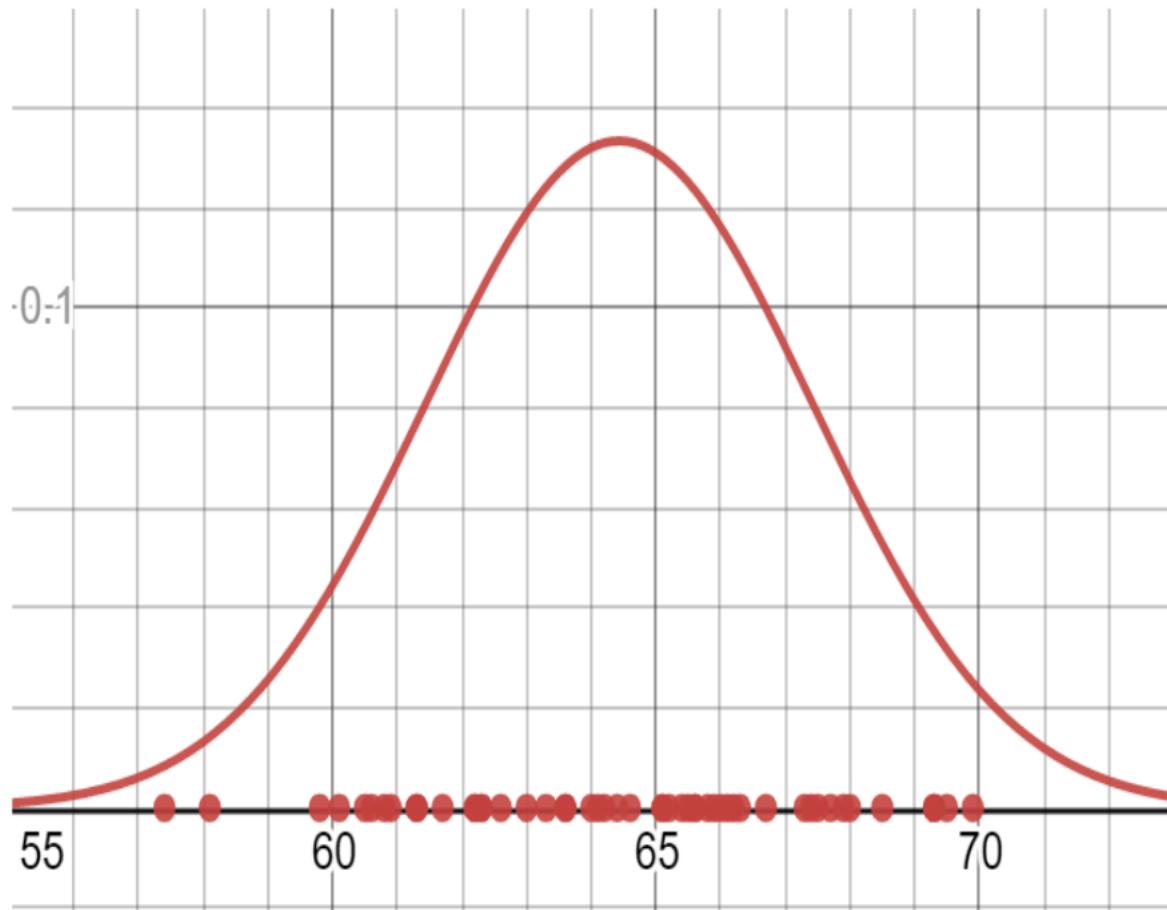


Figure 3-6. A normal distribution fitted to data points

Looking at the bell curve above, we can reasonably expect a golden retriever to have a weight most likely around 64.43 (the mean), but is unlikely at 55 or 73. Anything more extreme than that becomes virtually unlikely.

Properties of a Normal Distribution

The normal distribution has several important properties that make it useful:

- It's symmetrical, both sides are identically mirrored at the mean which is the center
- Most mass is at the center around the mean

- It has a spread (being narrow or wide) that is specified by standard deviation.
- The “tails” are the least likely outcomes, and approach zero infinitely but never touch zero.
- It resembles a lot of phenomena in nature and daily life, and even generalizes non-normal problems because of the central limit theorem which we will talk about shortly.

The Probability Density Function (PDF)

The standard deviation plays an important role in the normal distribution, because it defines how “spread out” it is. It is actually one of the parameters alongside the mean. The **probability density function (PDF)** that creates the normal distribution is as follows:

$$f(x) = \frac{1}{(\text{std dev}) * \sqrt{2\pi}} * e^{-\frac{1}{2} \left(\frac{x-\text{mean}}{\text{std dev}} \right)^2}$$

Wow that’s a mouthful, isn’t it? We even see our friend Euler’s Number e from Chapter 1 and some crazy exponents. Here is how we can express it in Python in **Example 3-9**.

Example 3-9. The normal distribution function in Python

```
# normal distribution, returns likelihood
def normal_pdf(x: float, mean: float, std_dev: float) -> float:
    return (1.0 / (2.0 * math.pi * std_dev ** 2) ** 0.5) *
        math.exp(-1.0 * ((x - mean) ** 2 / (2.0 * std_dev ** 2)))
```

There’s a lot to take apart here in this formula, but what’s important is that it accepts a mean and standard deviation as parameters, as well as an x value so you can look up the likelihood at that given value.

Just like the beta distribution in Chapter 2, the normal distribution is continuous. This means to retrieve a probability we need to integrate a range of x values to find an area.

In practice though, we will use SciPy to do these calculations for us.

Integrating the Normal Distribution

With the normal distribution, the vertical axis is not the probability but rather the likelihood for the data. To find the probability we need to look at a given range, and then find the area under the curve for that range. Let's say I want to find the probability of a golden retriever weighing between 62 and 66 pounds. Here is a visual in [Figure 3-7](#) of the range we want to find the area for.

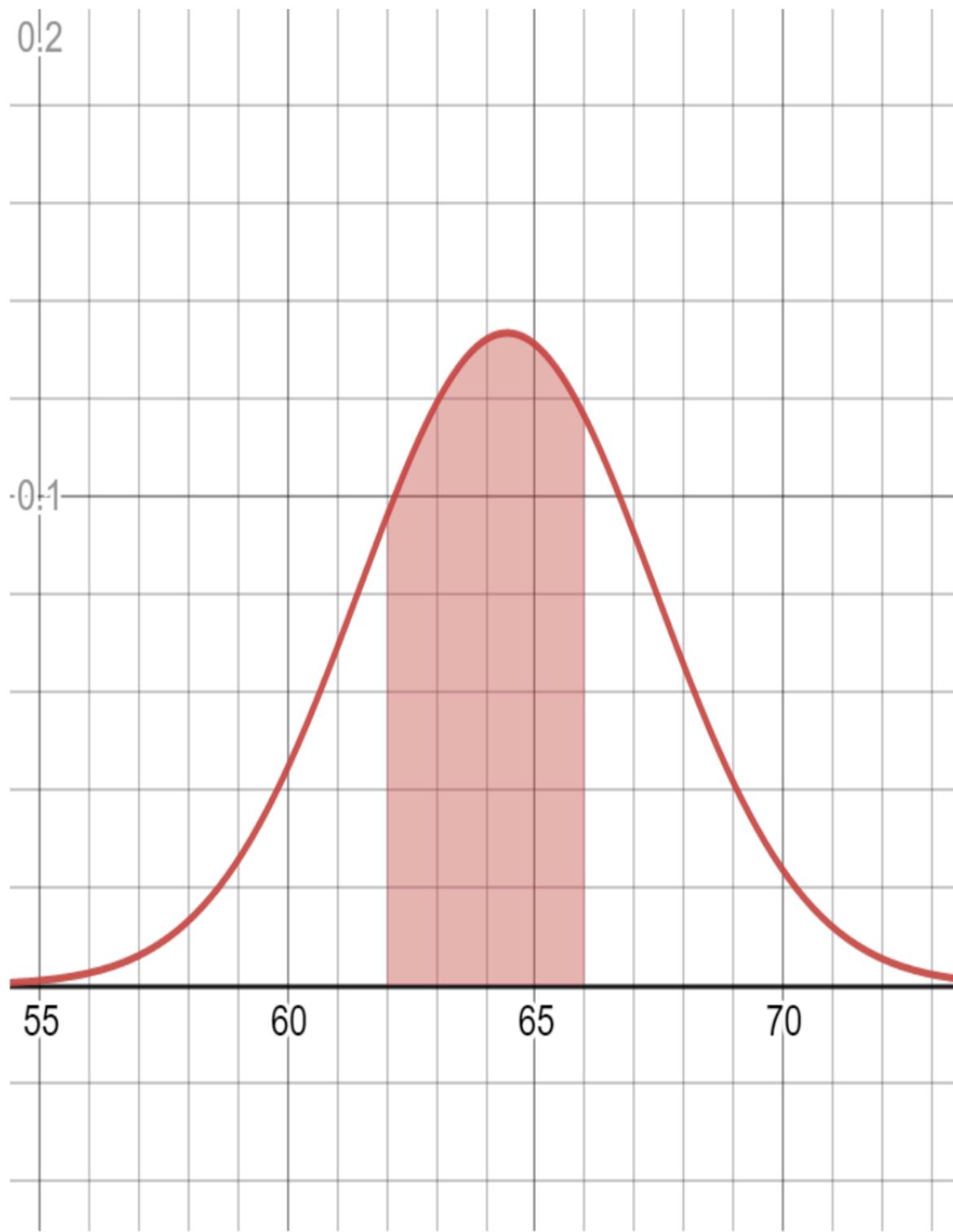


Figure 3-7. A cumulative density function (CDF) measuring probability between 62 and 66 pounds

We already did this task in Chapter 2 with the Beta distribution, and just like the Beta distribution there is a cumulative density function (CDF). Let's follow this approach.

Integrating using the Cumulative Density Function (CDF)

As we learned in the last chapter, the cumulative density function (CDF) provides the area *up to* a given x-value for a given distribution. Let's see what the CDF looks like for our golden retriever normal distribution, and put it alongside the PDF for reference in [Figure 3-8](#).

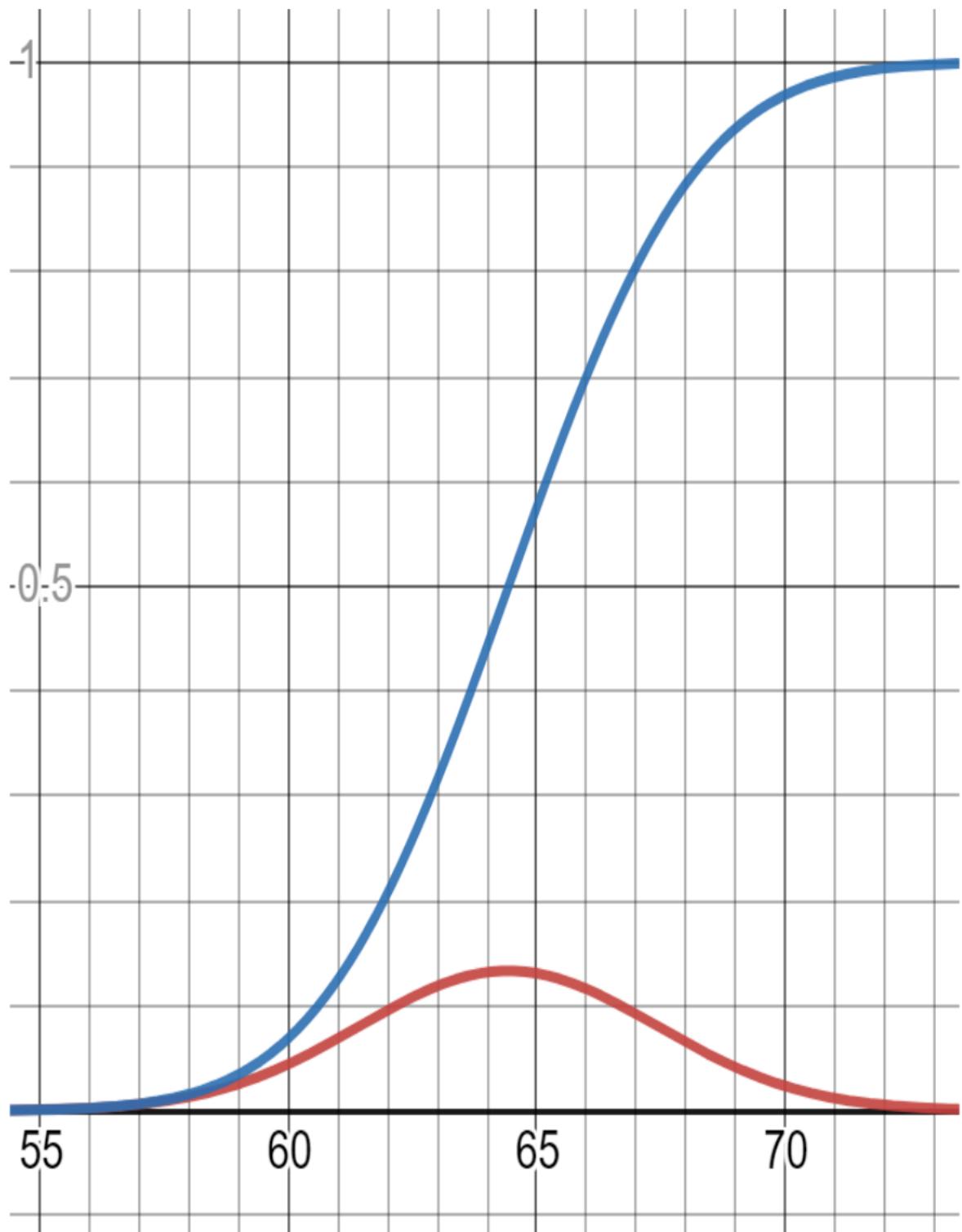


Figure 3-8. A probability density function (PDF) alongside its cumulative density function (CDF)

Notice there's a relationship between the two graphs. The CDF, which is an S-shaped curve (called a sigmoid curve), projects the area up to that range in the

PDF. Observe in [Figure 3-9](#) that when we capture the area from negative infinity up to 64.43 (the mean), our CDF shows a value of exactly .5 or 50%!

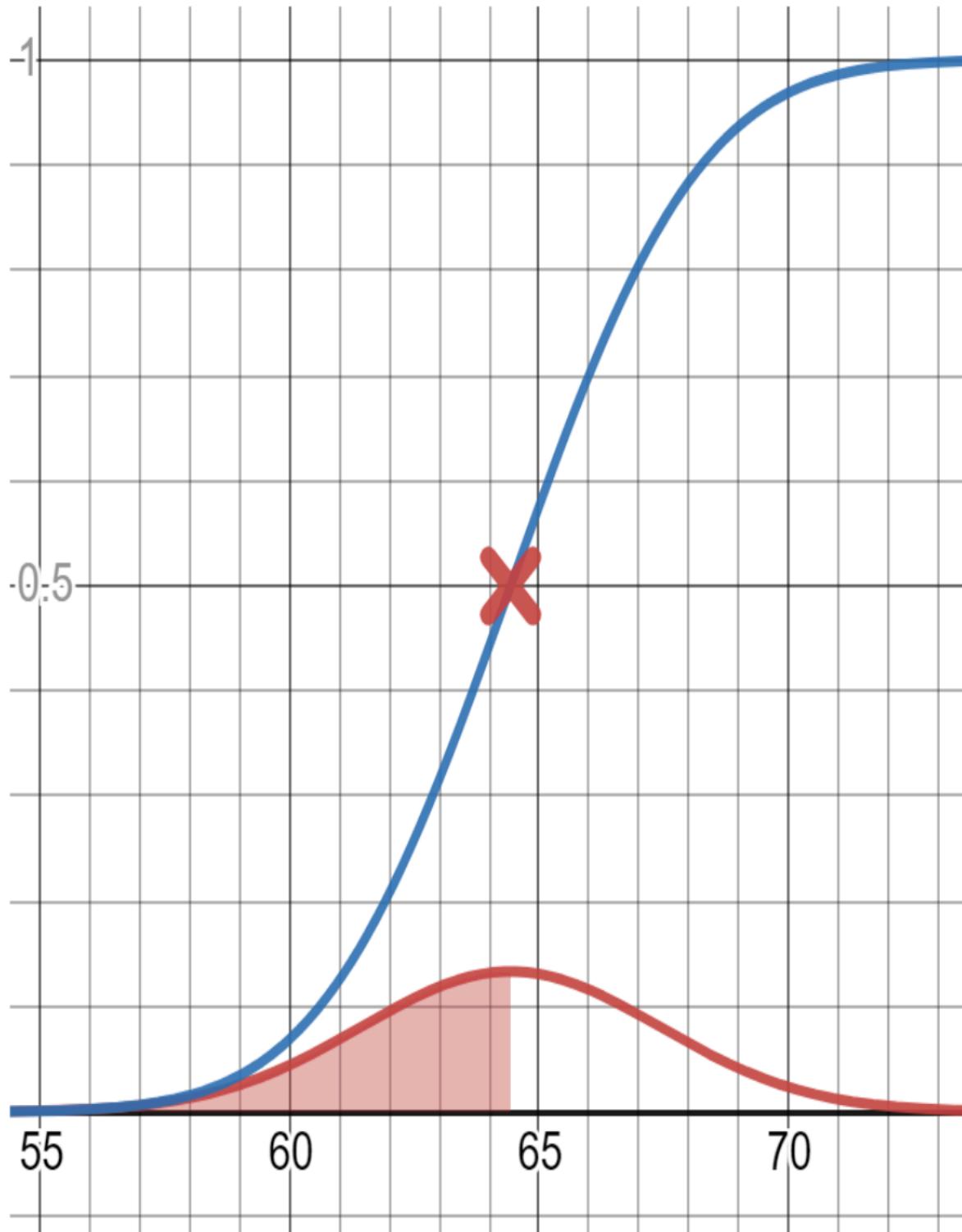


Figure 3-9. A PDF and CDF for golden retriever weights measuring probability up to the mean

This area of .50 or 50% up to the mean is known because of the symmetry of our normal distribution, and we can expect the other side of the bell curve to also have 50% of the area.

To calculate this area up to 64.43 in Python using SciPy, use the `norm.cdf()` function as shown in [Example 3-10](#).

Example 3-10. The normal distribution CDF in Python

```
from scipy.stats import norm

mean = 64.43
std_dev = 2.99

x = norm.cdf(64.43, mean, std_dev)

print(x) # prints 0.5
```

Just like we did in [Chapter 2](#), we can deductively find the area for a given range by subtracting areas. If we wanted to find the probability of observing a golden retriever between 62 and 66 pounds, we would calculate the area up to 66 and subtract the area up to 62 as visualized in [Figure 3-10](#).

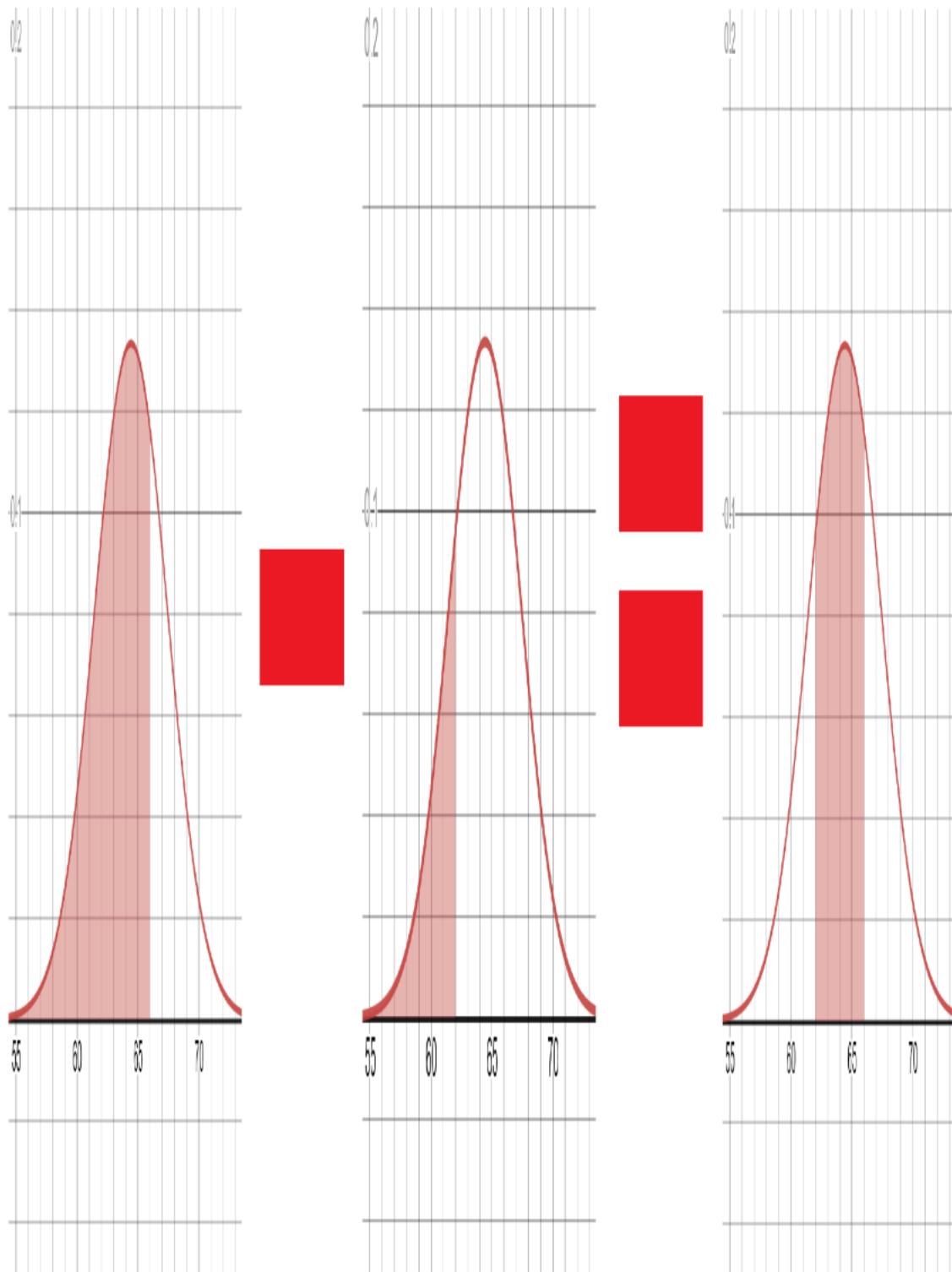


Figure 3-10. Finding a middle range of probability

Doing this in Python using SciPy is as simple as subtracting the two CDF operations shown in [Example 3-11](#).

Example 3-11. Getting a middle range probability using the CDF

```
from scipy.stats import norm

mean = 64.43
std_dev = 2.99

x = norm.cdf(66, mean, std_dev) - norm.cdf(62, mean, std_dev)

print(x) # prints 0.4920450147062894
```

You should find the probability of observing a golden retriever between 62 and 66 pounds to be 0.4920450147062894, or approximately 49.2%

The Inverse Cumulative Density Function (CDF)

When we start doing hypothesis testing later in this chapter, you will encounter situations where you need to look up an area on the CDF and then return the corresponding x-value. Of course this is a backwards usage of the CDF, so we will need to use the inverse CDF which flips the axes as shown in [Figure 3-11](#).

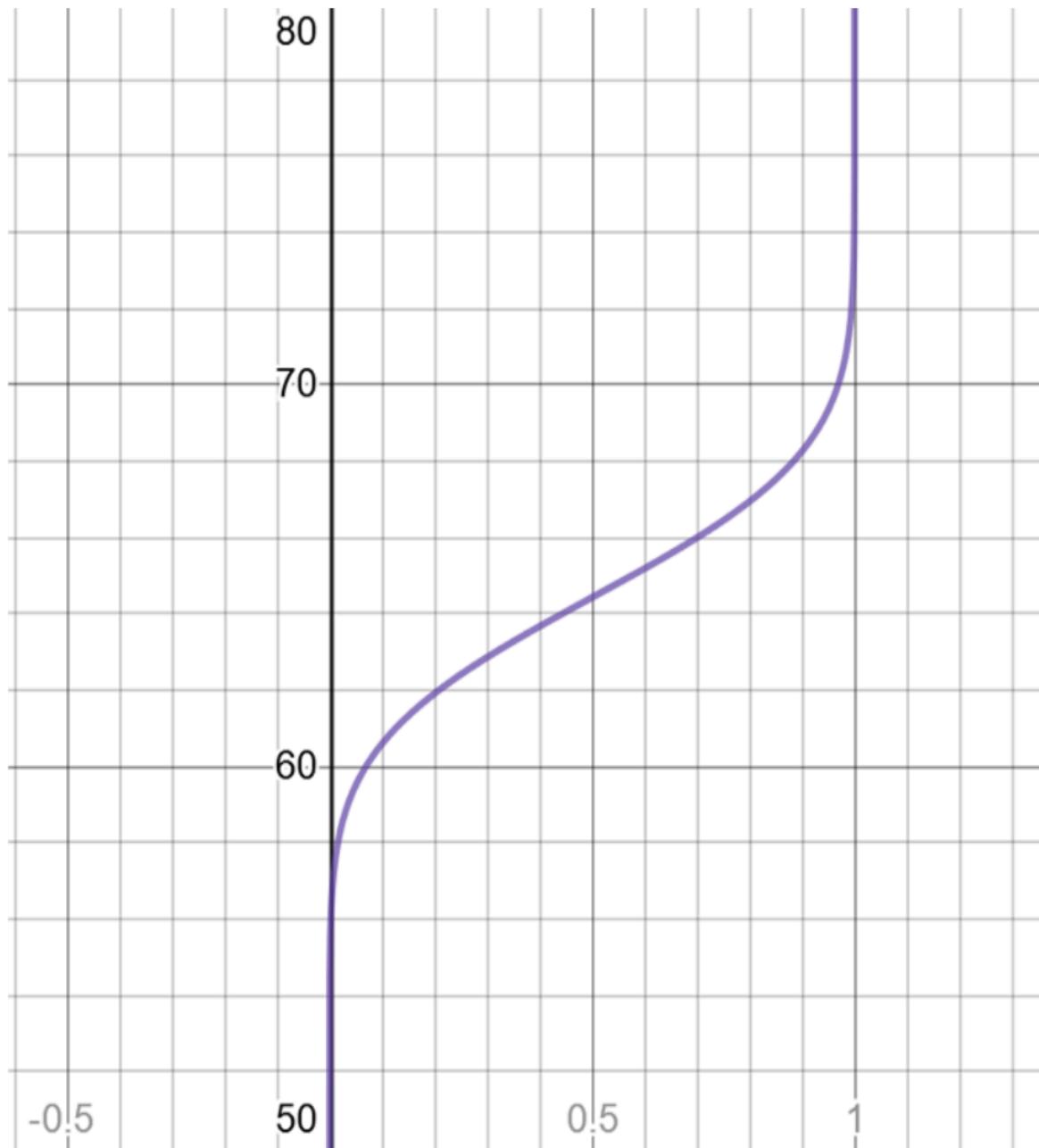


Figure 3-11. The inverse CDF, also called the PPF or quantile function

This way we can now look up a probability and then return the corresponding x-value, and in SciPy we would use the `norm.ppf()` function. For example, I want to find the weight that 95% of golden retrievers fall under. This is easy to do when I use the inverse CDF in [Example 3-12](#).

[Example 3-12. Using the inverse CDF \(called `ppf\(\)`\) in Python](#)

```
from scipy.stats import norm
```

```
x = norm.ppf(.95, loc=64.43, scale=2.99)
print(x) # 69.3481123445849
```

I find that 95% of golden retrievers are 69.348 pounds or less.

You can also use the inverse CDF to generate random numbers that follow the normal distribution. If I want to create a simulation that generates 1,000 realistic golden retriever weights, I just generate a random value between 0.0 and 1.0, pass it to the inverse CDF, and return the weight value as shown in [Example 3-13](#).

Example 3-13. Generating random numbers from a normal distribution

```
import random
from scipy.stats import norm

for i in range(0,1000):
    random_p = random.uniform(0.0, 1.0)
    random_weight = norm.ppf(random_p, loc=64.43, scale=2.99)
    print(random_weight)
```

Of course NumPy and other libraries can generate random values off a distribution for you, but this highlights one use case where the inverse CDF is handy.

CDF AND INVERSE CDF FROM SCRATCH

To learn how to implement the CDF and inverse CDF from scratch in Python, refer to Appendix A.

Z-Scores

It is common to re-scale a normal distribution so that the mean is 0 and the standard deviation is 1, which is known as the **standard normal distribution**. This allows us to make it easy to compare the spread of one normal distribution to another normal distribution, even if they have different means and variances.

Something that is of particular importance with the standard normal distribution is it expresses all x-values in terms of standard deviations, known as **Z-scores**. Turning an x-value into a Z-score uses a basic scaling formula.

$$z = \frac{x - \text{mean}}{\text{standard deviation}}$$

Here is an example. We have two homes from two different neighborhoods. Neighborhood A has a mean home value of \$140,000 and standard deviation of

\$3,000. Neighborhood B has a mean home value of \$800,000 and standard deviation of \$10,000.

$$\mu_A = 140,000$$

$$\mu_B = 800,000$$

$$\sigma_A = 3,000$$

$$\sigma_B = 10,000$$

Now we have two homes from each respective neighborhood. House A from neighborhood A is worth \$150,000 and House B from neighborhood B is worth \$815,000. Which home is more expensive relative to the average home in its neighborhood?

$$x_A = 150,000$$

$$x_B = 815,000$$

If we express these two values in terms of standard deviations, we can compare them relative to each neighborhood mean. Use the Z-score formula:

$$z = \frac{x - \text{mean}}{\text{standard deviation}}$$

$$z_A = \frac{150000 - 140000}{3000} = 3.\overline{333}$$

$$z_B = \frac{815000 - 800000}{10000} = 1.5$$

So the house in neighborhood A is actually much more expensive relative to its neighborhood than the house in neighborhood B, as they have values $3.\overline{333}$ and 1.5 respectively.

Here is how we can convert an x-value coming from a given distribution with a mean and standard deviation into a z-score, and vice versa, as shown in **Example 3-14**.

Example 3-14. Turn Z-scores in x values and vice versa

```
def z_score(x, mean, std):
    return (x - mean) / std
```

```

def z_to_x(z, mean, std):
    return (z * std) + mean

mean = 140000
std_dev = 3000
x = 150000

# Convert to Z-score and then back to X
z = z_score(x, mean, std_dev)
back_to_x = z_to_x(z, mean, std_dev)

print("Z-Score: {}".format(z)) # Z-Score: 3.333
print("Back to X: {}".format(back_to_x)) # Back to X: 150000.0

```

Above the `z_score()` function will take an x-value and scale it in terms of standard deviations, given a mean and standard deviation. The `z_to_x()` function takes a Z-score and converts it back to an x-value. Studying the two functions you can see their algebraic relationship, one solving for the Z-score and the other for the x-value. We then turn an x-value of 8.0 into a Z-score and then turn that Z-score back into an x-value.

COEFFICIENT OF VARIATION

A helpful tool related to measuring spread is the coefficient of variation. It compares two distributions and quantifies how spread out each of them are. It is simple to calculate: divide the standard deviation by the mean. Here is the formula alongside the example comparing two neighborhoods.

$$cv = \frac{\sigma}{\mu}$$

$$cv_A = \frac{3000}{140000} = 0.0214$$

$$cv_B = \frac{10000}{800000} = 0.0125$$

As seen above neighborhood A, while cheaper than neighborhood B, has more spread and therefore more price diversity than neighborhood B.

Inferential Statistics

Descriptive statistics, which we covered so far, is commonly understood. However when we get into inferential statistics the abstract relationships between sample and population comes into full play. These abstract nuances are not something you want to rush through, but rather take your time and absorb thoughtfully. As stated earlier, we are wired as humans to be biased and quickly come to conclusions. Being a good data science professional requires you to suppress that primal desire and consider the possibility that other explanations can exist. It is acceptable (perhaps even enlightened) to theorize there is no explanation at all and a finding is just coincidental and random.

First let's start with the theorem that lays the foundation for all inferential statistics.

The Central Limit Theorem

One of the reasons the normal distribution is useful is because it appears a lot in nature, such as adult golden retriever weights. However it shows up in a more fascinating context outside of natural populations. When we start measuring large enough samples from a population, even if that population does not follow a normal distribution, the normal distribution still makes an appearance.

Let's pretend I am measuring a population that is truly and uniformly random. Any value between 0.0 and 1.0 is equally likely, and no value has any preference. But something fascinating happens when we take increasingly large samples from this population, take the average of each, and then plot them into a histogram. Run this Python code in [Example 3-15](#) and observe the plot in [Figure 3-12](#).

Example 3-15. Exploring the central limit theorem in Python

```
# Samples of the uniform distribution will average out to a normal distribution.
import random
import plotly.express as px

sample_size = 31
sample_count = 1000

# Central limit theorem, 1000 samples each with 31 random numbers between 0.0 and 1.0
x_values = [(sum([random.uniform(0.0, 1.0) for i in range(sample_size)]) / sample_size)
             for _ in range(sample_count)]

y_values = [1 for _ in range(sample_count)]
```

```
px.histogram(x=x_values, y = y_values, nbins=20).show()
```

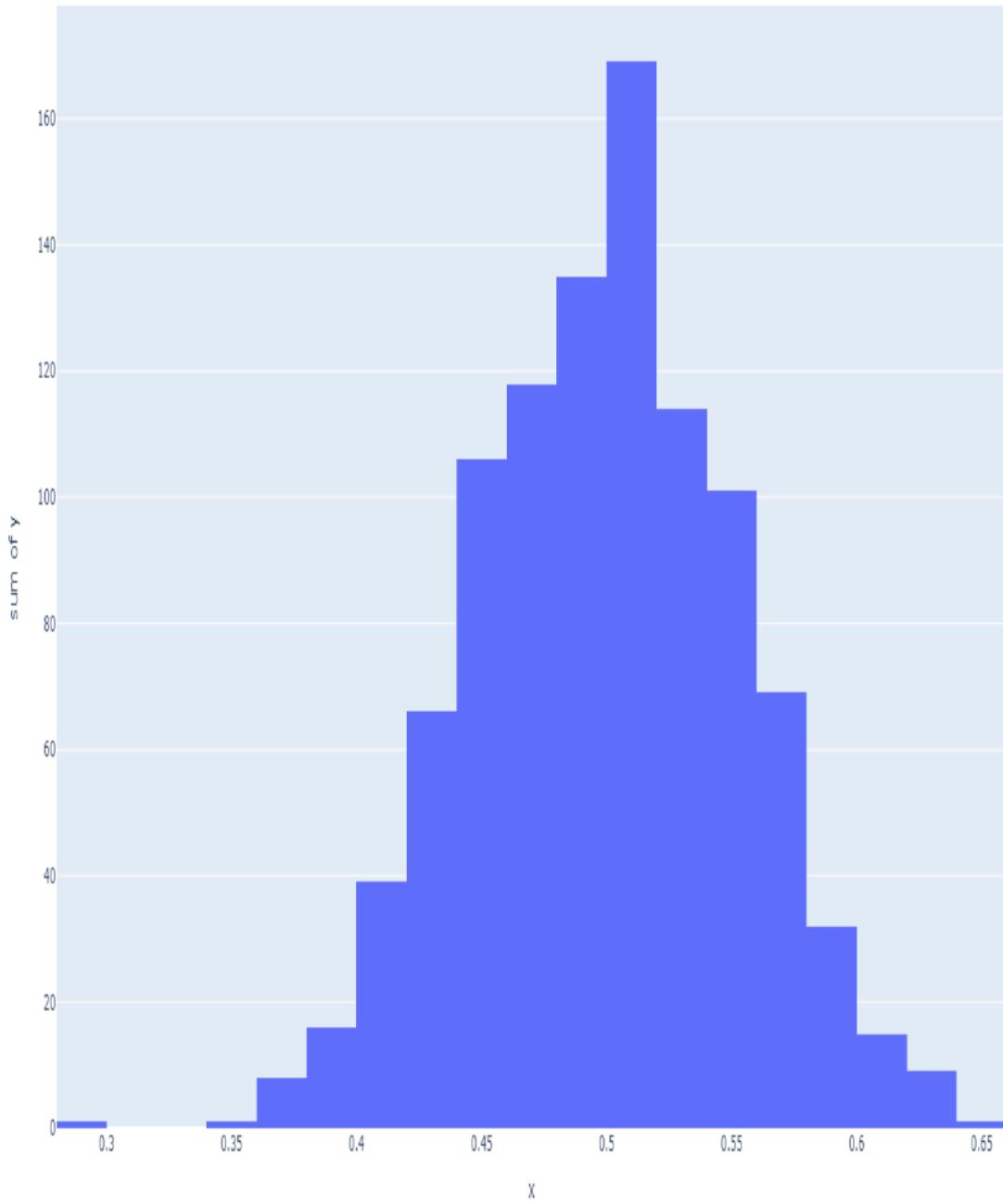


Figure 3-12. Taking the means of samples (each of size 31) and plotting them.

Wait how did uniformly random numbers, when sampled as groups of 31 and then averaged, roughly form a normal distribution? Any number is equally likely, right? Shouldn't the distribution be flat rather than bell-curved?

Here's what happening. The individual numbers in the samples alone will not create a normal distribution. The distribution will be flat where any number is equally likely (known as a **uniform distribution**). But when we group them as samples and average them they form a normal distribution.

This is because of the **central limit theorem**, which states interesting things happen when we take large enough samples of a population, calculate the mean of each, and plot them as a distribution:

1. The mean of the sample means is equal to the population mean.
2. If the population is normal, then the sample means will be normal.
3. If the population is not normal, but the sample size is greater than 30, roughly the sample means will still form a normal distribution.
4. The standard deviation of the sample means equals the population standard deviation divided by the square root of n .

$$\text{sample standard deviation} = \frac{\text{population standard deviation}}{\sqrt{\text{sample size}}}$$

Why is all the above important? These behaviors allows us to infer useful things about populations based on samples, even for non-normal populations. If you modify the code above and try smaller sample sizes of 1 or 2, you will not really see a normal distribution emerge. But as you approach 31 or more, you will see that we converge onto a normal distribution as shown in [Figure 3-13](#).

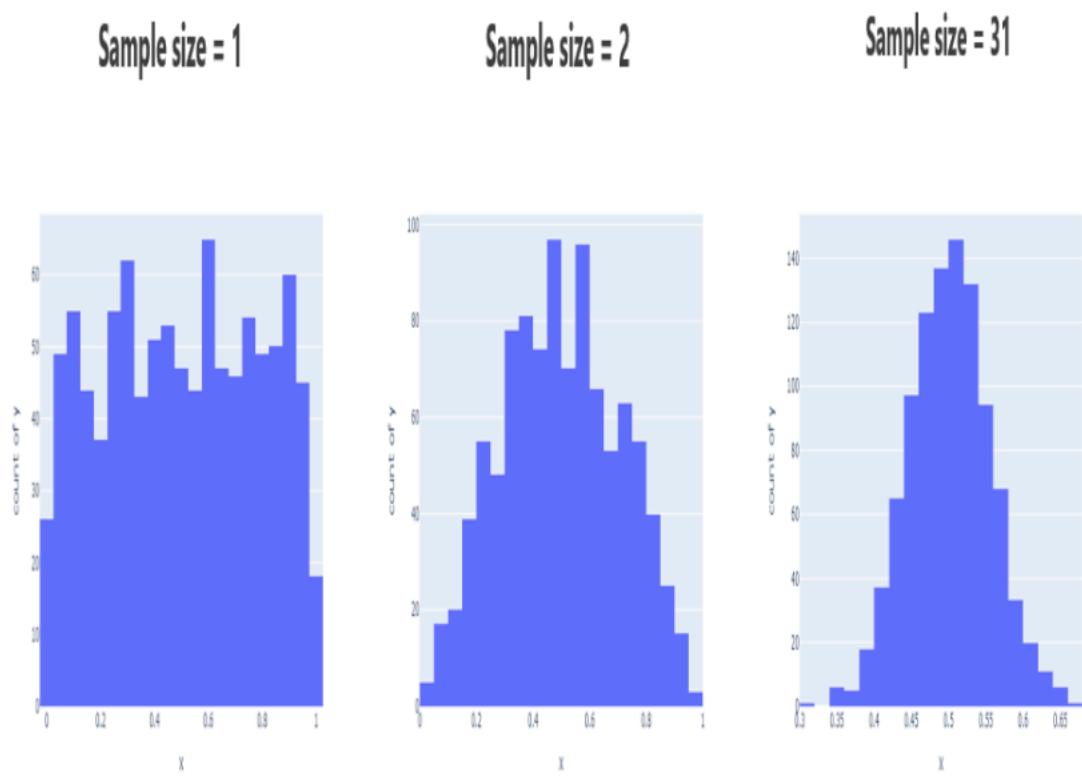


Figure 3-13. Larger sample sizes approach the normal distribution

31 is the textbook number in statistics because that is when our sample distribution often converges onto the population distribution, particularly when we measure sample means or other parameters. When you have less than 31 items in your sample, that is when you have to rely on the T-Distribution rather than the normal distribution, which has increasingly fatter tails the smaller your sample size is. We will briefly talk about this later, but first let's assume we have at least 31 items in our samples when we talk about confidence intervals and testing.

HOW MUCH SAMPLE IS ENOUGH?

While 31 is the textbook number of items you need in a sample to satisfy the central limit theorem and see a normal distribution, this sometimes is not the case. There are cases you will need an even larger sample, such as when the underlying distribution is asymmetrical or multimodal (meaning it has several peaks rather than one at the mean).

In summary, having larger samples are better when you are uncertain of the underlying probability distribution. You can read more [in this article](#).

Confidence Intervals

You may have heard the term confidence interval, which often confuses statistics newcomers and students. A **confidence interval** is a range calculation showing how confidently we believe a sample mean (or other parameter) falls in a range for the population mean.

Base on a sample of 31 golden retrievers with a sample mean of 64.408 and a sample standard deviation of 2.05, I am 95% confident that the population mean lies between 63.686 and 65.1296. How do I know this? Let me show you and if you get confused, circle back to this paragraph and remember what we are trying to achieve. I highlighted that in bold for a reason!

I first start out by choosing a **level of confidence (LOC)**, which will contain the desired probability for the population mean range. I want to be 95% confident that my sample mean falls in the population mean range I will calculate. That's my LOC. We can leverage the central limit theorem and infer what this range for the population mean is. First, I need the **critical Z value** which is the symmetrical range in a standard normal distribution that gives me 95% probability in the center as highlighted in [Figure 3-14](#).

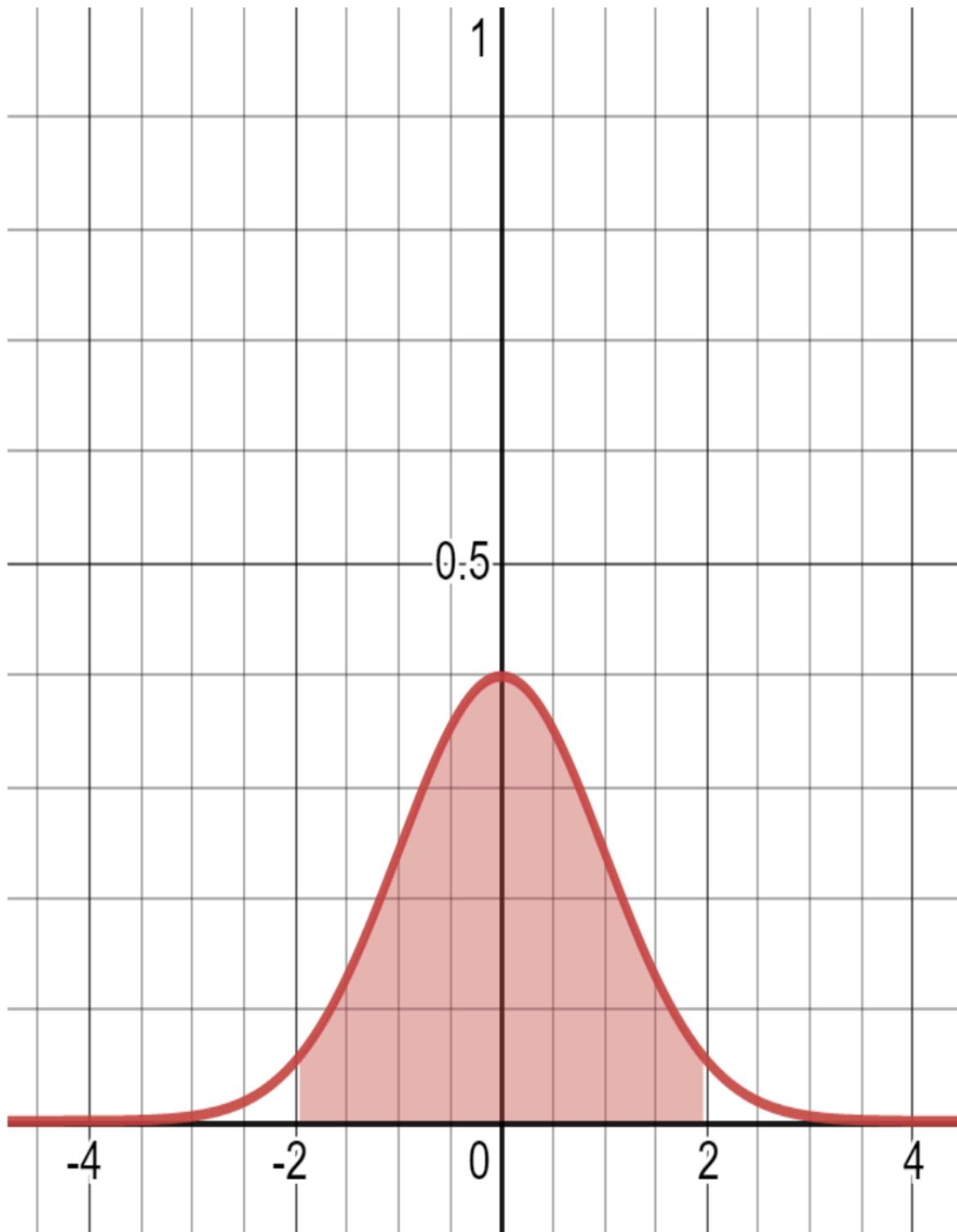


Figure 3-14. 95% symmetrical probability in the center of a standard normal distribution

How do we calculate this symmetrical range containing .95 of the area? It's easier to grasp as a concept than as a calculation. You may instinctively want to use the CDF, but then you may realize there's a few more moving parts here.

First you need to leverage the inverse CDF. Logically to get 95% of the symmetrical area in the center, we would chop off the tails which have the remaining 5% of area. Splitting that remaining 5% area in half would give us 2.5% area in each tail. Therefore, the areas we want to look up the x-values for are .025 and .975 as shown in [Figure 3-15](#).

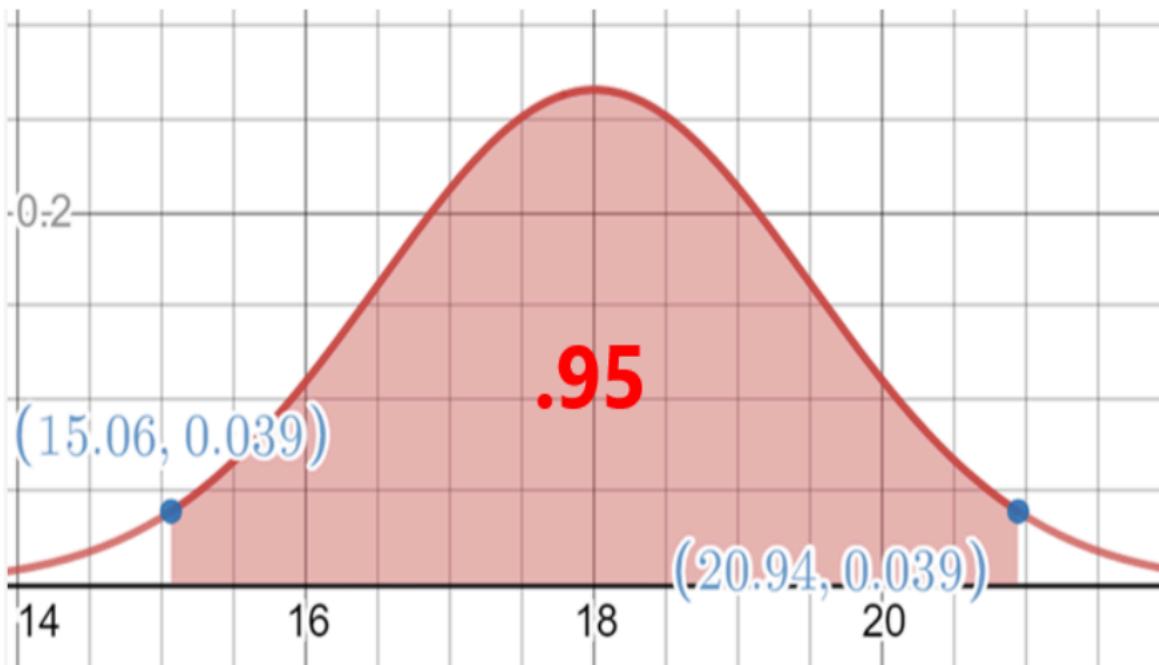


Figure 3-15. We want the x-values that give us areas .025 and .975

We can look up the x-value for area .025, and the x-value for area .975, and that will give us our center range containing 95% of the area. We will then return the corresponding lower and upper z-values containing this area (remember, we are using standard normal distribution here!). Let's calculate this in Python as shown in [Example 3-16](#).

Example 3-16. Retrieving a critical Z-value

```
from scipy.stats import norm

def critical_z_value(p):
    norm_dist = norm(loc=0.0, scale=1.0)
    left_tail_area = (1.0 - p) / 2.0
    upper_area = 1.0 - ((1.0 - p) / 2.0)
    return norm_dist.ppf(left_tail_area), norm_dist.ppf(upper_area)

print(critical_z_value(p=.95))
# (-1.959963984540054, 1.959963984540054)
```

Okay so we get ± 1.95996 which is our critical Z-value capturing 95% of probability at the center of the standard normal distribution. Next I'm going to leverage the central limit theorem to produce the **margin of error (E)**, which is the range around the sample mean that contains the population mean at that level of confidence. Recall that our sample of 31 golden retrievers has a mean of 64.408 and standard deviation of 2.05. The formula to get this margin of error is as shown:

$$E = \pm z_c \frac{s}{\sqrt{n}}$$

$$E = \pm 1.95996 * \frac{2.05}{\sqrt{31}}$$

$$E = \pm 0.72164$$

If we apply that margin of error against the sample mean, we finally get the confidence interval!

$$95\% \text{ confidence interval} = 64.408 \pm 0.72164$$

Here is how we calculate this confidence interval in Python from beginning to end in [Example 3-17](#).

Example 3-17. Calculating a confidence interval in Python

```
from math import sqrt
from scipy.stats import norm

def critical_z_value(p):
    norm_dist = norm(loc=0.0, scale=1.0)
    left_tail_area = (1.0 - p) / 2.0
    upper_area = 1.0 - ((1.0 - p) / 2.0)
    return norm_dist.ppf(left_tail_area), norm_dist.ppf(upper_area)

def confidence_interval(p, sample_mean, sample_std, n):
    # Sample size must be greater than 30

    lower, upper = critical_z_value(p)
    lower_ci = lower * (sample_std / sqrt(n))
    upper_ci = upper * (sample_std / sqrt(n))

    return sample_mean + lower_ci, sample_mean + upper_ci
```

```
print(confidence_interval(p=.95, sample_mean=64.408, sample_std=2.05, n=31))  
# (63.68635915701992, 65.12964084298008)
```

So the way to interpret this is “based on my sample of 31 golden retriever weights with sample mean 64.408 and sample standard deviation of 2.05, I am 95% confident the population mean lies between 63.686 and 65.1296”. That is how we describe our confidence interval.

One interesting thing to note here too is that in our margin of error formula, the larger n becomes the narrower our confidence interval becomes! This makes sense because if we have a larger sample, we are more confident in the population mean falling in a smaller range, hence why it’s called a confidence interval.

One caveat to put here is that for this to work, our sample size must be at least 31 items. This goes back to the central limit theorem and if we want to apply a confidence interval to a smaller sample, we need to use a distribution with higher variance (fatter tails reflecting more uncertainty). This is what the T-distribution is for and we will visit that at the end of this chapter.

In [Chapter 5](#) we will continue to use confidence intervals for linear regressions.

Understanding P-Values

When we say something is *statistically significant*, what do we mean by that? We hear it used loosely and frequently but what does it mathematically mean?

Technically, it has to do with something called the p-value which is a hard concept for many folks to grasp. But I think the concept of p-values makes more sense when you trace it back to its invention. While this is an imperfect example, it gets across some big ideas.

In 1925, a mathematician named Ronald Fisher was at a party. One of his colleagues Muriel Bristol claimed she could detect when tea was poured before milk simply by tasting it. Intrigued by the claim, Ronald set up an experiment on the spot.

He prepared 8 cups of tea. 4 had milk poured first, the other 4 had tea poured first. He then presented them to his connoisseur colleague and asked her to identify the pour order for each. Remarkably she identified them all correctly, and the probability of this happening by chance is 1 in 70, or 0.01428571.

This 1.4% probability is what we call the **p-value**, the probability something occurring by chance rather than because of a hypothesized explanation. Without going down a rabbit hole of combinatorial math, the probability that Muriel completely guessed the cups correctly is 1.4%. What exactly does that tell you?

When we frame an experiment, whether it is determining if organic donuts cause weight gain or living near power lines causes cancer, we always have to entertain the possibility that random luck played a role. Just like there is a 1.4% chance Muriel identified the cups of tea correctly simply by guessing, there's always a chance randomness just gave us a good hand like a slot machine. This helps us frame our **null hypothesis (H0)**, saying that the variable in question had no impact on the experiment and any positive results is just random luck. The **alternative hypothesis (H1)** poses that a variable in question (called the **controlled variable**) is causing a positive result.

Traditionally the threshold for statistical significance is a p-value of 5% or less, or .05. Since .014 is less than .05, this would mean we can reject our null hypothesis that Muriel was randomly guessing. We can then promote the alternative hypothesis that Muriel has a special ability to detect whether tea or milk was poured first.

Now one thing this tea party example did not capture is that when we calculate a p-value, we capture all probability of that event or rarer. We will address this as we dive into the next example using the normal distribution.

Hypothesis Testing

Past studies have shown that the mean recovery time for a cold is 18 days, with a standard deviation of 1.5 days, and follows a normal distribution.

This means there is approximately 95% chance of recovery taking between 15 and 21 days as shown in [Figure 3-16](#) and [Example 3-18](#).

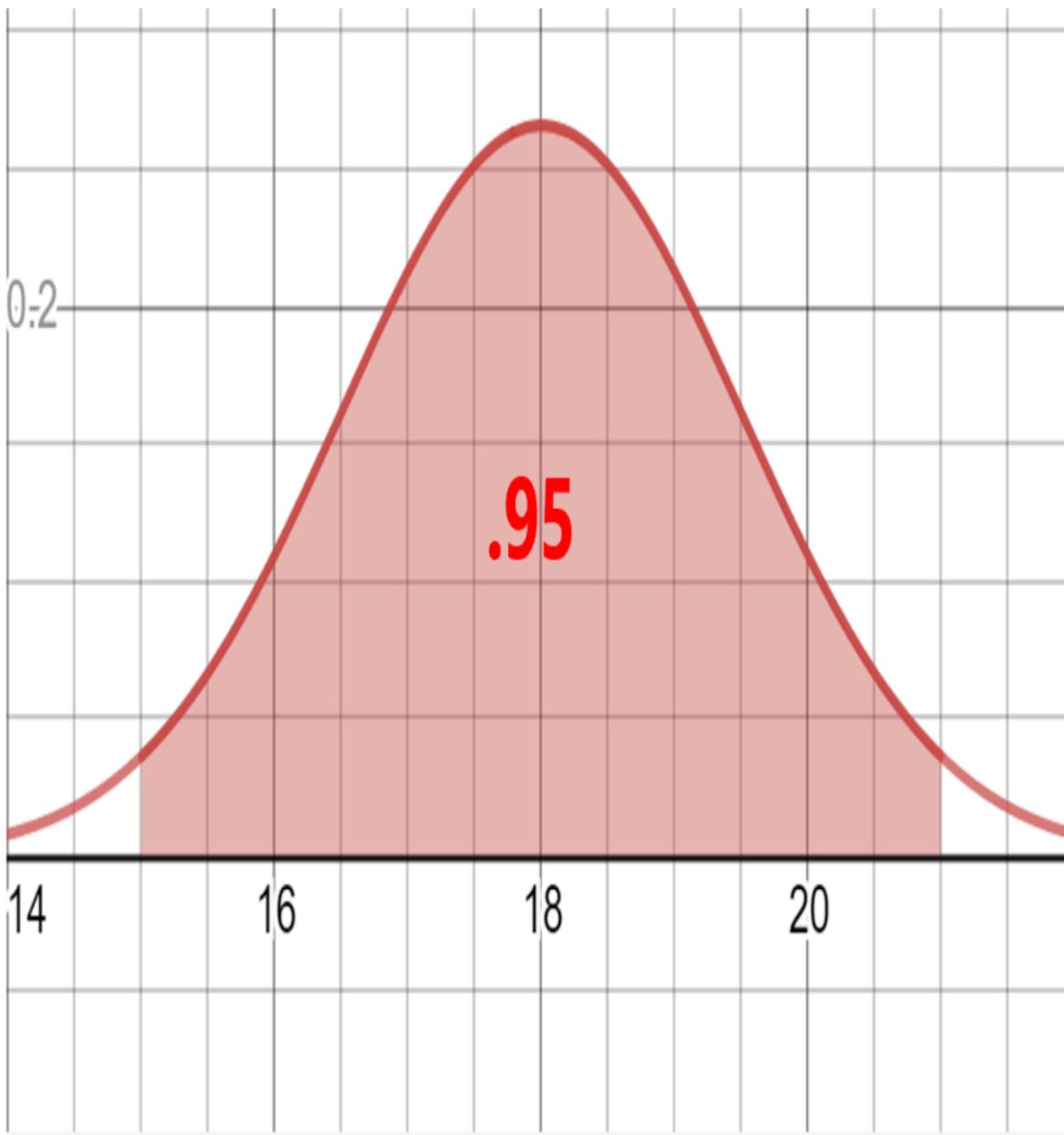


Figure 3-16. There is 95% chance of recovery between 15 and 21 days

Example 3-18. Calculating the probability of recovery between 15 and 21 days

```
from scipy.stats import norm

# Cold has 18 day mean recovery, 1.5 std dev
mean = 18
std_dev = 1.5

# 95% probability recovery time takes between 15 and 21 days.
x = norm.cdf(21, mean, std_dev) - norm.cdf(15, mean, std_dev)

print(x) # 0.9544997361036416
```

We can infer then from the remaining 5% probability that there's a 2.5% chance of recovery taking longer than 21 days and a 2.5% chance of less than 15 days. Hold onto that bit of information because it will be critical later! That drives our p-value.

Now let's say an experimental new drug was given to a test group of 40 people and it took an average of 16 days for them to recover from the cold as shown in [Figure 3-17](#).

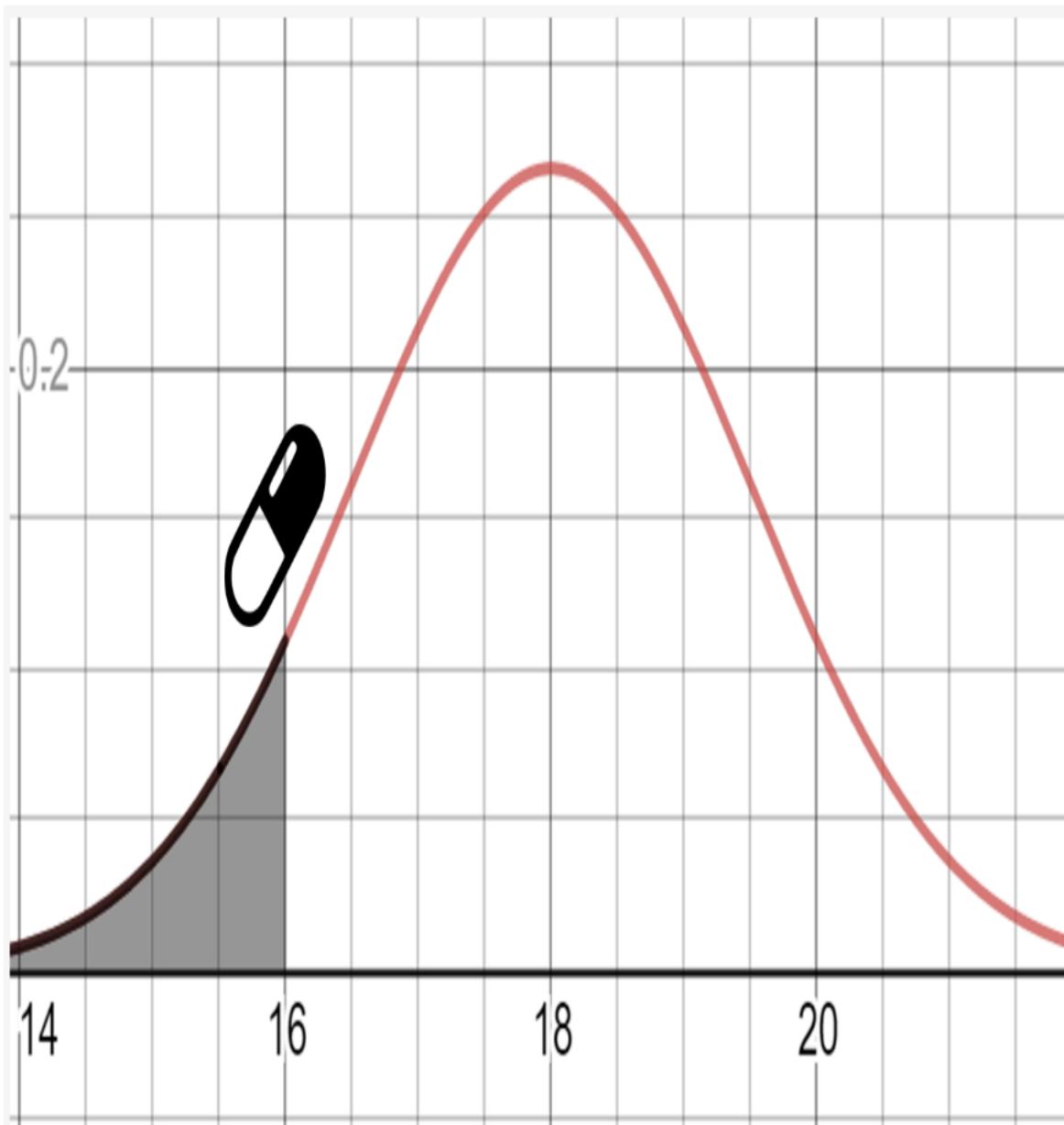


Figure 3-17. A group taking a drug took 16 days to recover

Did the drug have an impact? If you reason long enough, you may realize what we are asking is this: does the drug show a statistically significant result? Or did the drug not work and the 16-day recovery was a coincidence with the test group? That first question frames our alternative hypothesis, while the second question frames our null hypothesis.

There are two ways we can calculate this: the One-tailed and two-tailed test. We will start with the one-tailed.

One-Tailed Test

When we approach the **one-tailed test**, we typically frame our null and alternative hypotheses using inequalities. We hypothesize around the population mean and say that it either is greater than/equal to 18 (the null hypothesis H_0) or less than 18 (the alternative hypothesis H_1).

$$H_0 : \text{population mean} \geq 18$$

$$H_1 : \text{population mean} < 18$$

To reject our null hypothesis, we need to show that our sample mean (of the patients who took the drug) is not likely to have been coincidental. Since a P-value of .05 or less is traditionally considered statistically significant, we will use that as our threshold (Figure 3-17). When we calculate this in Python using the inverse CDF as shown in [Figure 3-18](#) and [Example 3-19](#), we find that approximately 15.51 is the number of recovery days that gives us .05 area on the left tail.

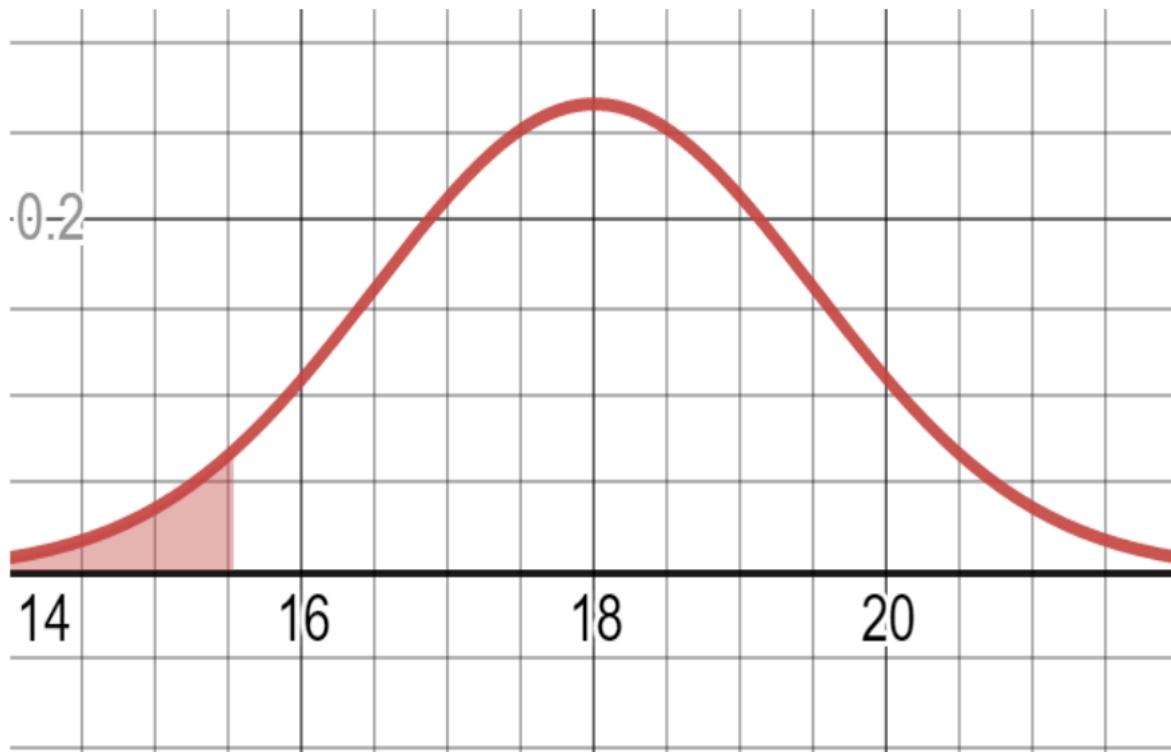


Figure 3-18. Getting the x-value with 5% of area behind it

Example 3-19. Python code for getting x-value with 5% of area behind it

```
from scipy.stats import norm

# Cold has 18 day mean recovery, 1.5 std dev
mean = 18
std_dev = 1.5

# What x-value has 5% of area behind it?
x = norm.ppf(.05, mean, std_dev)

print(x) # 15.53271955957279
```

Therefore if we achieve an average 15.53 or less days of recovery time in our sample group, our drug is considered statistically significant enough to have shown an impact. However, our sample mean of recovery time is actually 16 days and does not fall into this null hypothesis rejection zone. Therefore the statistical significance test has failed as shown in [Figure 3-19](#).

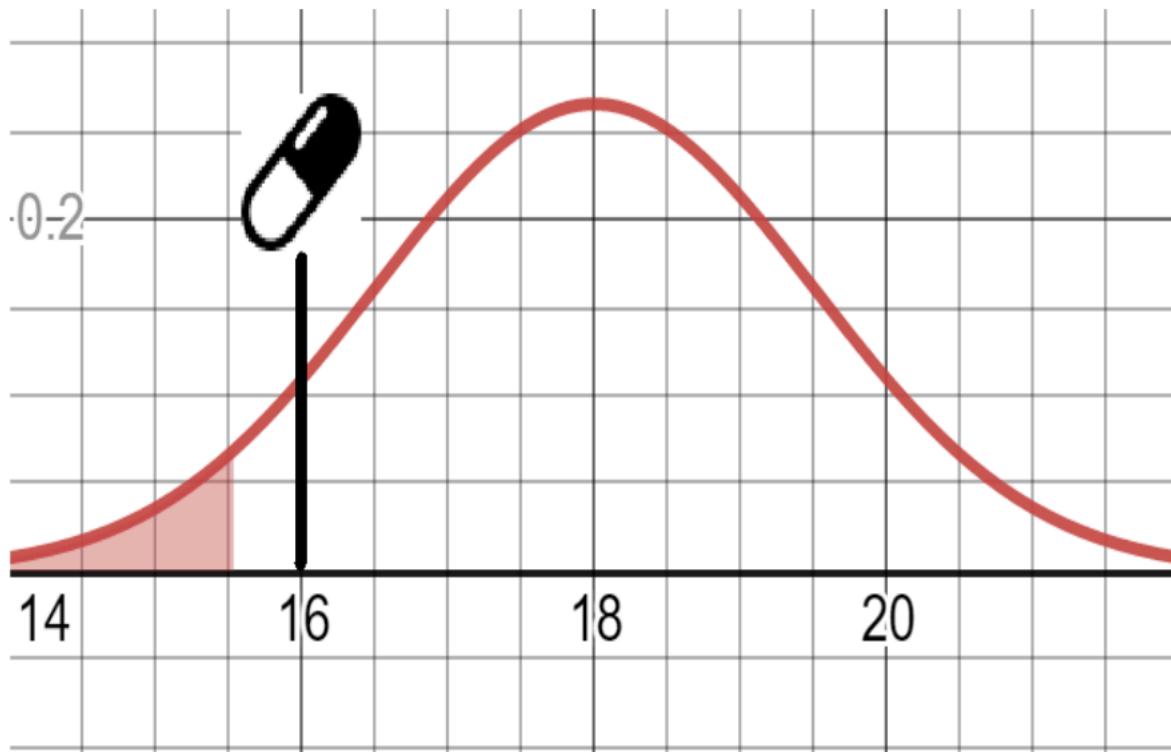


Figure 3-19. We have failed to prove our drug test result is statistically significant

The area up to that 16-day mark is our p-value, which is .0912 and we calculate it in Python as shown in [Example 3-20](#).

Example 3-20. Calculating the 1-tailed p-value

```
from scipy.stats import norm

# Cold has 18 day mean recovery, 1.5 std dev
mean = 18
std_dev = 1.5

# Probability of 16 or less days
p_value = norm.cdf(16, mean, std_dev)

print(p_value) # 0.09121121972586788
```

Since the p-value of .0912 is greater than our statistical significance threshold of .05, we do not consider the drug trial a success and fail to reject our null hypothesis.

Two-Tailed Test

The previous test we performed is called the one-tailed test because it only looks for statistical significance on one tail. However it is often safer and better practice

to use a two-tailed test. We will elaborate why later.

To do a **two-tailed test**, we frame our null and alternative hypothesis in an “equal” and “not equal” structure. In our drug test, we will say the null hypothesis has a mean recovery time of 18 days. But our alternative hypothesis is the mean recovery time is not 18 days thanks to the new drug.

$$H_0 : \text{population mean} = 18$$

$$H_1 : \text{population mean} \neq 18$$

This has an important implication. We are structuring our alternative hypothesis to not test whether the drug improves cold recovery time, but if it had *any* impact. This includes testing if it increased the duration of the cold. Is this helpful? Hold that thought.

Naturally this means we spread our p-value statistical significance threshold to both tails, not just one. If we are testing for a statistical significance of 5% then we split it and give each 2.5% half to each tail. If our drug mean recovery time falls in either region our test is successful and we reject the null hypothesis ([Figure 3-20](#)).

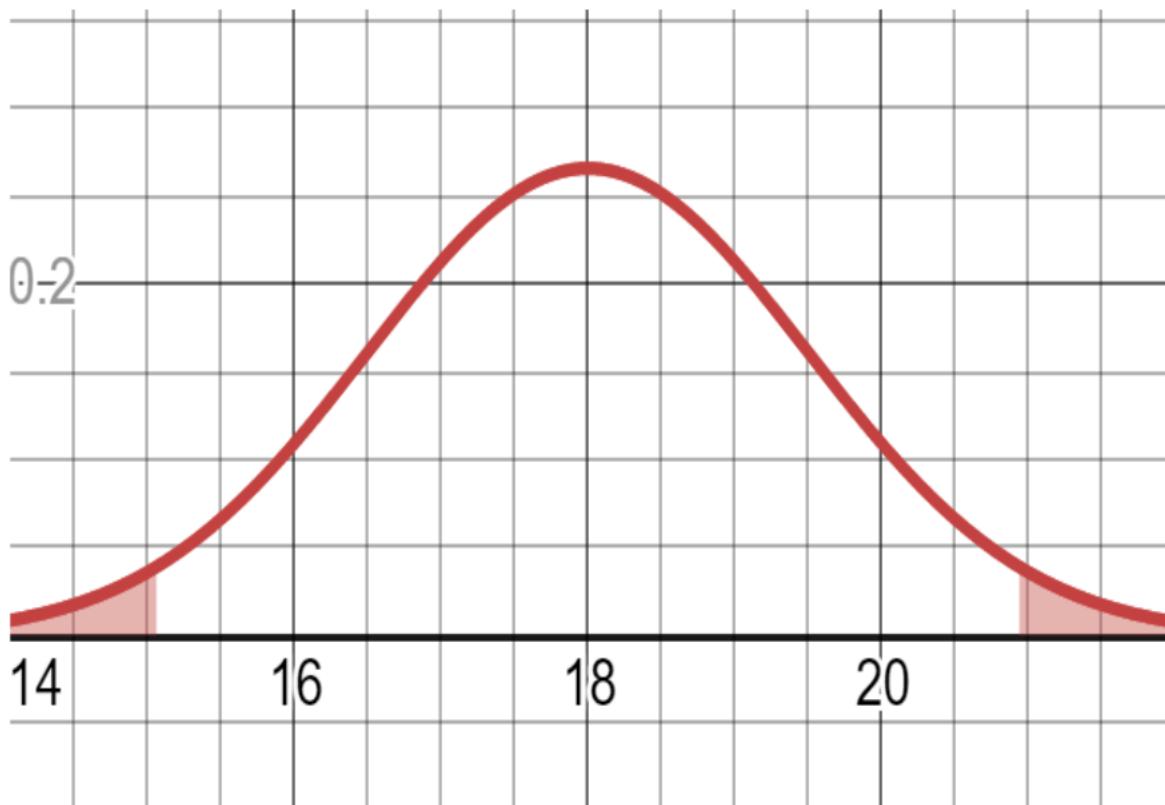


Figure 3-20. A 2-tailed test

The x-values for the lower tail and upper tail are 15.06 and 20.93, meaning if we are under or over respectively we reject the null hypothesis. Here are those two values calculated using the inverse CDF below in [Figure 3-21](#) and [Example 3-21](#). Remember, to get the upper tail we take .95 and then add the .025 piece of significance threshhold to it, giving us .975.

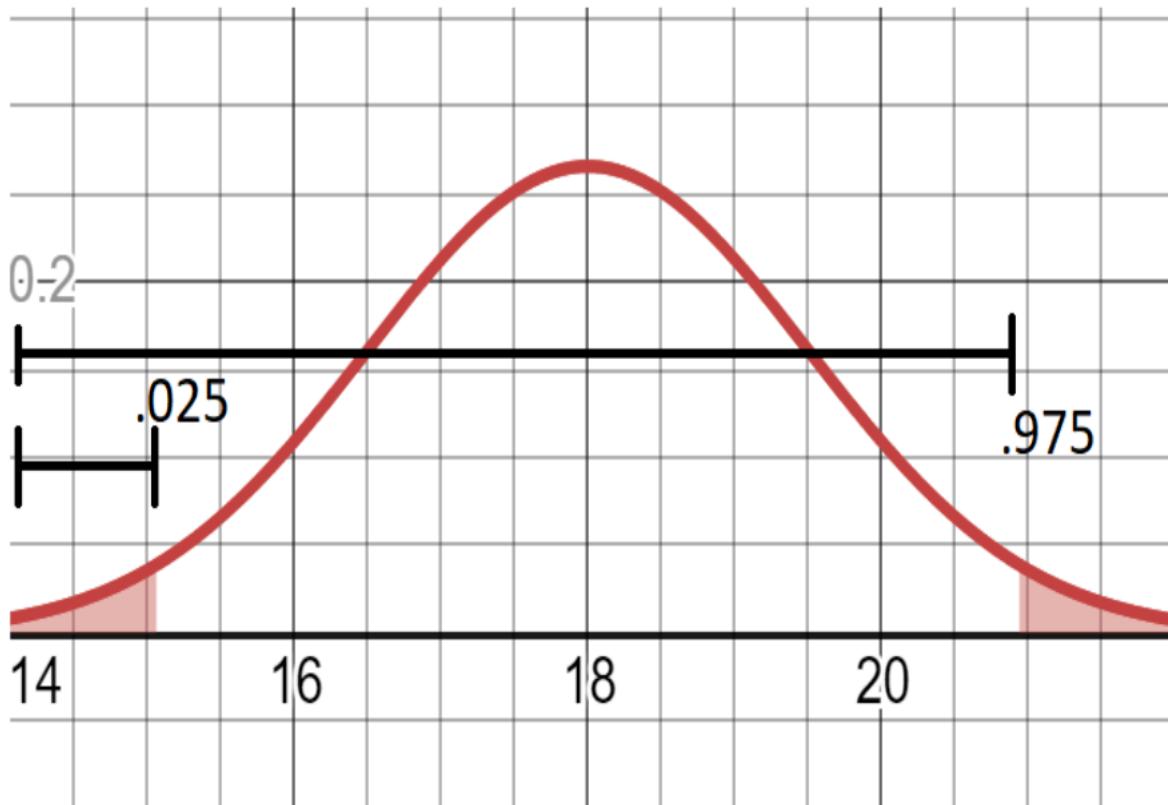


Figure 3-21. Calculating the central 95% of normal distribution area

Example 3-21.

```
from scipy.stats import norm

# Cold has 18 day mean recovery, 1.5 std dev
mean = 18
std_dev = 1.5

# What x-value has 2.5% of area behind it?
x1 = norm.ppf(.025, mean, std_dev)

# What x-value has 97.5% of area behind it?
x2 = norm.ppf(.975, mean, std_dev)

print(x1) # 15.060054023189918
print(x2) # 20.93994597681008
```

The sample mean value for the drug test group is 16, and 16 is not less than 15.06 nor greater than 20.9399. So like the 1-tailed test we still fail to reject the null hypothesis. Our drug still has not shown any statistical significance to have any impact as shown in [Figure 3-22](#).

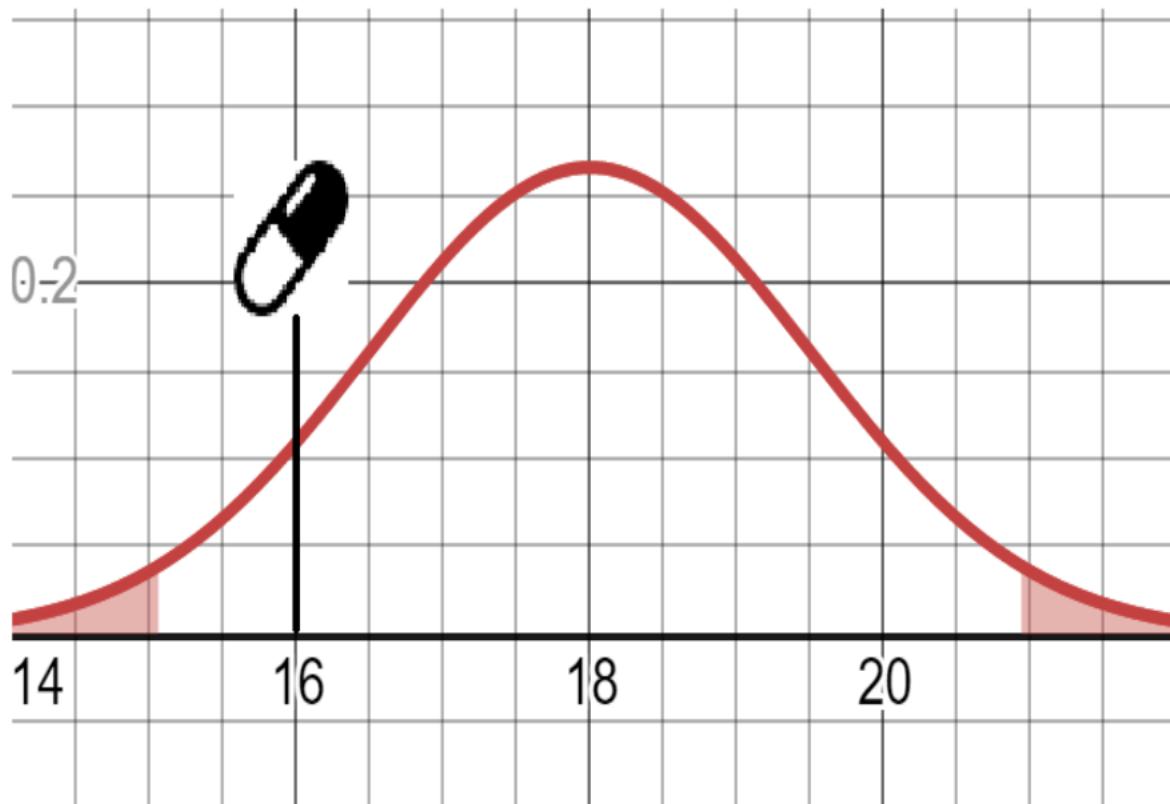


Figure 3-22. The 2-tailed test has failed to prove statistical significance

But what is the p-value? This is where it gets interesting with 2-tailed tests. Our p-value is going to not just capture the area to the left of 16, but also the symmetrical equivalent area on the right tail as well. Since 16 is 4 days below the mean, we will also capture the area above 20 which is 4 days above the mean ([Figure 3-23](#)). This is capturing the probability of an event or rarer, on both sides of the bell curve.

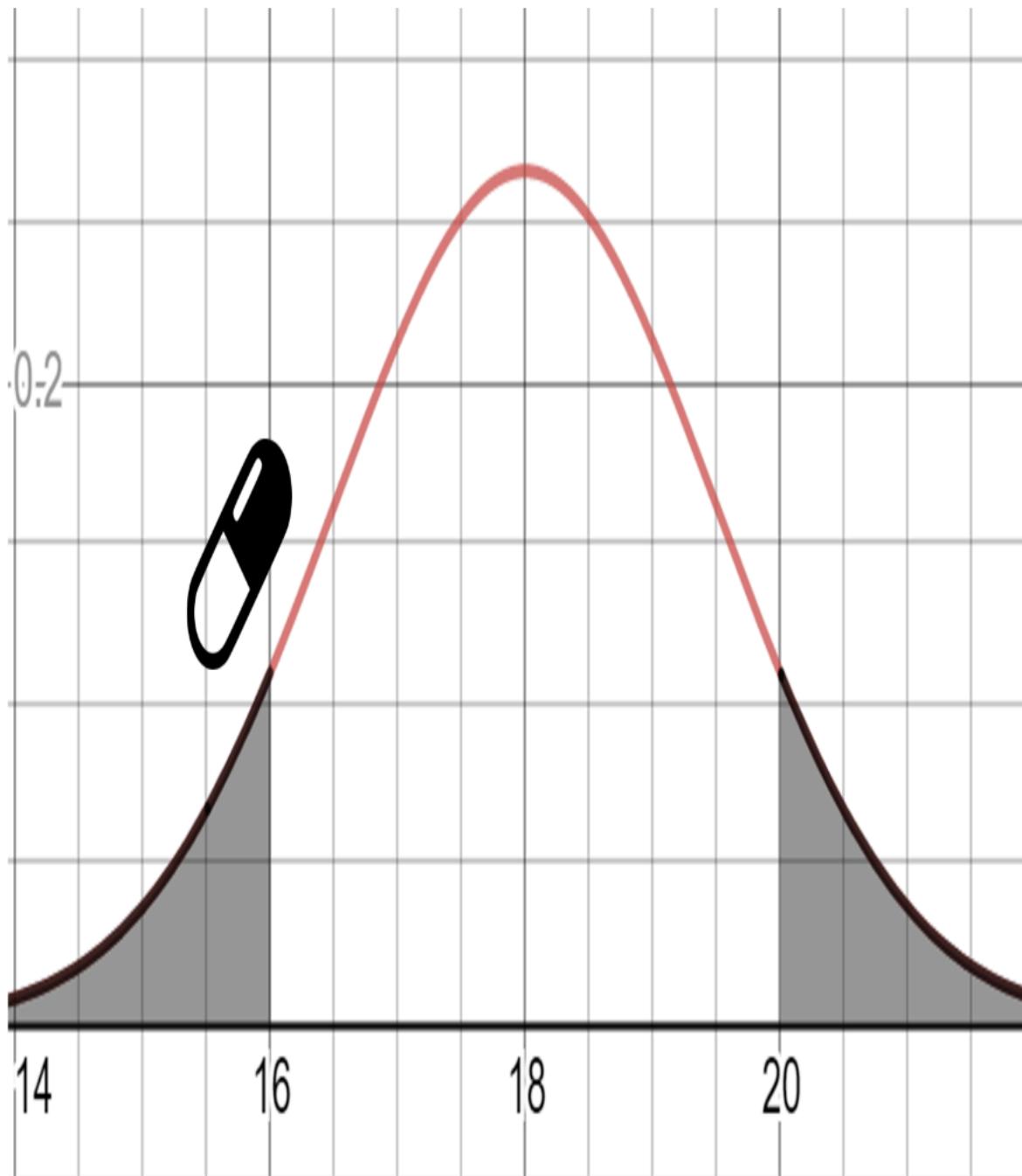


Figure 3-23. The p-value adds symmetrical sides for statistical significance

When we sum both those areas, we get a p-value of .1824. This is a lot greater than .05, so it definitely does not pass our p-value threshold of .05 ([Example 3-22](#)).

Example 3-22. Calculating the 2-tailed p-value

```
from scipy.stats import norm
```

```

# Cold has 18 day mean recovery, 1.5 std dev
mean = 18
std_dev = 1.5

# Probability of 16 or less days
p1 = norm.cdf(16, mean, std_dev)

# Probability of 20 or more days
p2 = 1.0 - norm.cdf(20, mean, std_dev)

# P-value of both tails
p_value = p1 + p2

print(p_value) # 0.18242243945173575

```

So why do we also add the symmetrical area on the opposite side in a 2-tailed test? This may not be the most intuitive concept, but first remember how we structured our hypotheses:

$$H_0 : \text{population mean} = 18$$

$$H_1 : \text{population mean} \neq 18$$

If we are testing in an “equals 18” versus “not equals 18” capacity, we have to capture any probability that is of equal or less value on both sides. After all we are trying to prove significance, and that includes anything that is equally or less likely to happen. We did not have this special consideration with the 1-tailed test that used only “greater/less than” logic. But when we are dealing with “equals/not equals” our interest area goes in both directions..

So what are the practical implications of the 2-tailed test? How does it affect whether we reject the null hypothesis? Ask yourself this: which one sets a higher threshold? You will notice that even when our objective is to show we may have lessened something (the cold recovery time using a drug), reframing our hypothesis to show any impact (greater or lesser) creates a higher significance threshold. If our significance threshold is a p-value of .05 or less, our 1-tailed test was closer to acceptance at p-value .0912 as opposed to the 2-tailed test which was about double that at p-value .182.

This means the 2-tailed test makes it harder to reject the null hypothesis and demands stronger evidence to pass a test. Also think of this: what if our drug could worsen colds and make them last longer? It may be helpful to capture that probability too and account for variation in that direction. This is why 2-tailed

tests are preferable in most cases. They tend to be more reliable and not bias the hypothesis in just one direction.

We will use hypothesis testing and P-values again in [Chapter 5](#) and Chapter 6.

BEWARE OF P-HACKING!

There is a growing problem in the scientific research community called p-hacking, where researchers shop for statistically significant p-values of .05 or less. This is not hard to do with “big data”, machine learning, and data mining where we can traverse hundreds or thousands of variables and then find statistically significant (but coincidental) relationships between them.

Why do so many researchers p-hack? Many are probably not aware they are doing it. It’s easy to keep tuning and tuning a model, omitting “noisy” data and changing parameters, until it gives the “right” result. Others are simply under pressure by academia and industry to produce lucrative, not objective, results.

If you are hired by the Calvin Cereal Company to study whether Frosted Chocolate Sugar Bombs causes diabetes, do you expect to give an honest analysis and get hired again? What if a manager asks you to produce a forecast showing \$15 million in sales next quarter for a new product launch? You have no control over what sales will be, but you’re asked for a model that produces a pre-determined result. In the worst case, you may even be held accountable when it proves wrong. Unfair, but it happens!

This is why soft skills like diplomacy can make a difference in a data science professional’s career. If you can share difficult and inconvenient stories in a way that productively gets an audience, that is a tremendous feat. Just be mindful of the organization’s managerial climate and always propose alternate solutions. If you end up in a no-win situation where you are asked to p-hack, and could be held accountable for when it backfires, then definitely change work environments!

The T-Distribution: Dealing with Small Samples

Let’s briefly address how to deal with smaller samples of 30 or less, and we will need this when we do linear regression in [Chapter 5](#). Whether we are calculating confidence intervals or doing hypothesis testing, if we have 30 or less items in a sample we would opt to use a T-Distribution instead of a normal distribution. The **T-distribution** is like a normal distribution but has fatter tails to reflect more variance and uncertainty. [Figure 3-24](#) shows a normal distribution (dashed) alongside a T-distribution with 1 degree of freedom (solid):

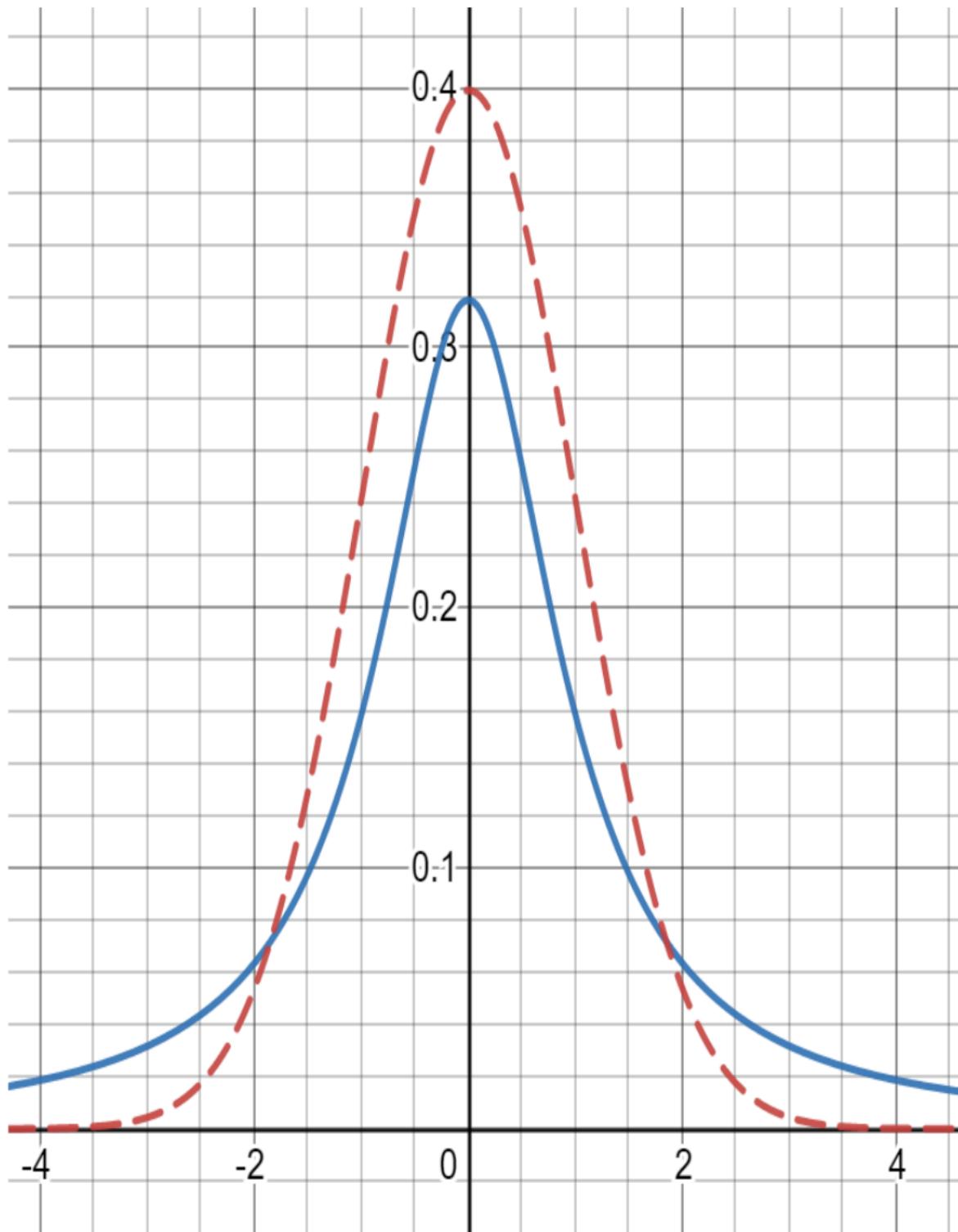


Figure 3-24. The T-Distribution alongside a normal distribution. Note the fatter tails.

The smaller the sample size is, the fatter the tails get in a T-distribution. But what's interesting is after you approach 31 items the T-distribution is nearly

indistinguishable from the normal distribution, which neatly reflects the ideas behind the central limit theorem.

Example 3-23 below shows how to find the **critical t-value** for 95% confidence. You can use this for confidence intervals and hypothesis testing when you have a sample size of 30 or less. It's conceptually the same as the critical z-value, but we are using a T-distribution instead of a normal distribution to reflect greater uncertainty. The smaller the sample size, the larger the range reflecting greater uncertainty.

Example 3-23.

```
from scipy.stats import t

# get critical value range for 95% confidence
# with a sample size of 25

n = 25
lower = t.ppf(.025, df=n-1)
upper = t.ppf(.975, df=n-1)

print(lower, upper)
-2.063898561628021 2.0638985616280205
```

Note that `df` is the “degrees of freedom” parameter, and as outlined earlier it should be 1 less of the sample size.

BEYOND THE MEAN

We can use confidence intervals and hypothesis testing to measure other parameters besides the mean, including variance/standard deviation as well as proportions (e.g. 60% of people report improved happiness after walking 1 hour a day). These require other distributions such as the Chi-Square distribution instead of the normal distribution. As stated, these are all beyond the scope of this book but hopefully you have a strong conceptual foundation to extend into these areas if needed.

We will however use confidence intervals and hypothesis testing throughout **Chapter 5** and Chapter 6.

Big Data Considerations and Texas Sharpshooter Fallacy

One final thought before we close this chapter. As we have discussed, randomness plays such a role in validating our findings and we always have to account for its possibility. Unfortunately with “big data”, machine learning, and other data-mining tools the scientific method has suddenly become a practice done backwards. This can be precarious and allow me to demonstrate why, adapting an example from Gary Smith in his book *Standard Deviations*.

Let’s pretend I draw 4 playing cards from a fair deck. There’s no game or objective here other than to draw 4 cards and observe them. I get 2 tens, a three, and a two. “This is interesting” I say. “I got 2 tens, a three, and a two. Is this meaningful? Are the next four cards I draw also going to be two consecutive numbers and a pair? What’s the underlying model here?”

See what I did there? I took something that was completely random and I not only looked for patterns, but I tried to make a predictive model out of them. What has subtly happened here is I never made it my objective to get these four cards with these particular patterns. I observed them *after* they occurred.

This is exactly what data mining falls victim to every day: finding coincidental patterns in random events. With huge amounts of data and fast algorithms looking for patterns, it’s not hard to find things that look meaningful but actually are just random coincidences.

This is analogous to me firing a gun at a wall. I then draw a target around the hole and I then bring my friends over to show off my amazing marksmanship. Silly, right? Well many people in data science figuratively do this everyday and it is known as the **Texas Sharpshooter Fallacy**. They set out to act without an objective, stumble on something rare, and then point out what they found somehow creates predictive value.

The problem is the law of truly large numbers says rare events are likely to be found, we just do not know which ones. When we encounter rare events, we highlight and even speculate what might have caused them. The probability of a specific person winning the lottery is highly unlikely, but yet someone *is* going to win the lottery. Why should we be surprised when there is a winner? Unless

somebody predicted the winner, nothing meaningful happened other than a random person got lucky.

This also applies to correlations which we will study in [Chapter 5](#). With an enormous dataset with thousands of variables, is it hard to find statistically significant findings with a .05 p-value? You betcha! I'll find thousands of those. I'll even show evidence <http://www.tylervigen.com/spurious-correlations> [the number of Nicholas Cage movies correlates with the number of pool drownings in a year].

So to prevent the Texas Sharpshooter Fallacy and falling victim to big data fallacies, try to use structured hypothesis testing and gather data for that objective. If you utilize data mining, try to obtain fresh data to see if your findings still hold up. Finally, always consider the possibility that things can be coincidental and if there is not a common sense explanation then it probably was coincidental.

We learned how to hypothesize before gathering data, but data mining gathers data then hypothesizes. Ironically, we are often more objective starting with a hypothesis, because we then seek out data to deliberately prove and disprove our hypothesis.

Conclusions

We learned a lot in this chapter, and you should feel good about getting this far. This was probably one of the harder topics in this book! We learned not only descriptive statistics from the mean to the normal distribution, but we also tackled confidence intervals and hypothesis testing.

Hopefully you see data a little differently. It is snapshots of something rather than a complete capturing of its context. Data on its own is not very useful, and we need context, curiosity, and analysis of where it came from before we can make meaningful insights with it. We covered how to describe data as well as infer attributes about a larger population based on a sample. Finally we addressed some of the data mining fallacies we can stumble into if we are not careful, and how to remedy that with fresh data and common sense.

Do not feel bad if you need to go back and review some of the content in this chapter, because there's a lot to digest. It is also important to get in the hypothesis testing mindset if you want to be successful at a data science and machine learning

career. Few practitioners take time to link statistics and hypothesis testing concepts to machine learning, and it unfortunately has slowed progress in the field substantially. Understandability and explainability is the next frontier of machine learning, so continue to learn and integrate these ideas as you progress throughout the rest of this book and the rest of your career.

Exercises

1. You bought a spool of 1.75mm filament for your 3D printer. You want to measure how close the filament diameter really is to 1.75 mm. You use a caliper tool and sample the diameter five times on the spool:
1.78, 1.75, 1.72, 1.74, 1.77
Calculate the mean and standard deviation for this set of values.
2. A manufacturer says the Z-Phone smart phone has a mean consumer life of 42 months with a standard deviation of 8 months. Assuming a normal distribution, what is the probability a given random Z-Phone will last between 20 and 30 months?
3. I am skeptical that my 3D printer filament is not 1.75mm in average diameter as advertised. I sampled 34 measurements with my tool. The sample mean is 1.715588 and the sample standard deviation is 0.029252.
What is the 99% confidence interval for the mean of my entire spool of filament?
4. Your marketing department has started a new advertising campaign and wants to know if it affected sales, which in the past averaged \$10,345 a day with a standard deviation of \$552. The new advertising campaign ran for 45 days and averaged \$11,641 in sales.
Did the campaign affect sales? Why or why not? (Use a 2-tailed test for more reliable significance)

Chapter 4. Linear Algebra

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at thomasnield@live.com.

Changing gears a little bit, let's venture away from probability and statistics and into linear algebra. Sometimes people confuse linear algebra with basic algebra, thinking maybe it has to do with plotting lines using the algebraic function $y = mx + b$. This is why linear algebra should probably have been called “vector algebra” or “matrix algebra” because it is much more abstract. Linear systems play a role, but in a much more metaphysical way.

So, what exactly is linear algebra? Well, **linear algebra** concerns itself with linear systems, but represents them through vector spaces and matrices. If you do not know what a vector or a matrix is, do not worry! We will define and explore them in depth. Linear algebra is hugely fundamental to many applied areas of math, statistics, operations research, data science, and machine learning. When we work with data in any of these areas, we are using linear algebra and perhaps you may not even know it.

You can get away with not learning linear algebra for awhile, using machine learning and statistics libraries that do it all for you. But if we are going to get intuition behind these black boxes and be more effective at working with data, understanding the fundamentals of linear algebra is an inevitability.

Linear algebra is an enormous topic that can fill thick textbooks, so of course we cannot gain total mastery in just one chapter of this book. However, we can learn enough to be more comfortable with it and navigate the data science domain effectively. There will also be opportunities to apply it in later chapters of this book.

What is a Vector?

Simply put a **vector** is an arrow in space with a specific direction and length, often representing a piece of data. It is the central building block to linear algebra, including matrices and linear transformations. In its fundamental form, it has no concept of location so always imagine its tail starts at the origin of a Cartesian plane (0,0).

Figure 4-1 shows a vector \vec{v} that moves 3 steps in the horizontal direction and 2 steps in the vertical direction.

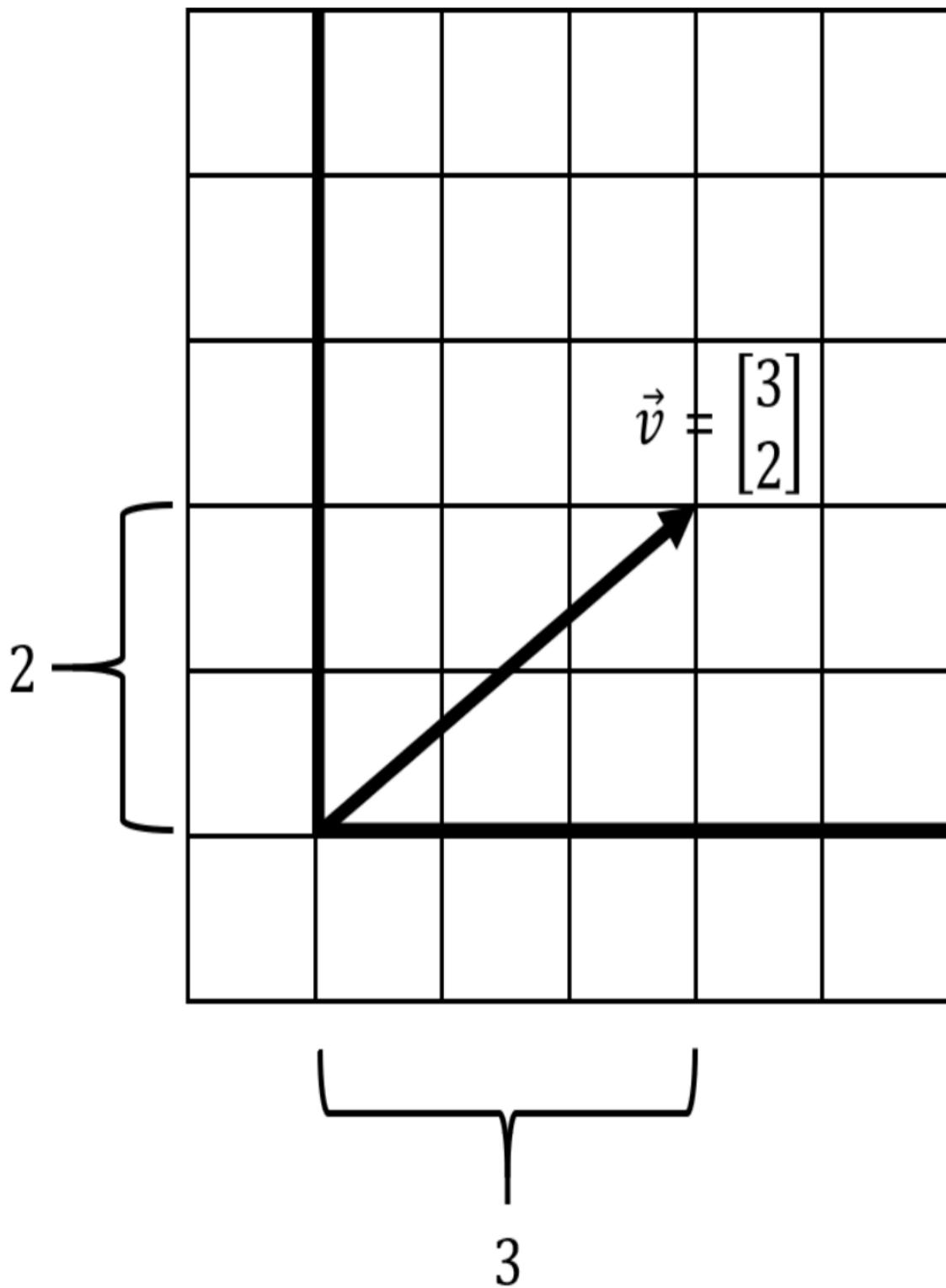


Figure 4-1. A simple vector

To emphasize again, the purpose of the vector is to visually represent a piece of data. If you have a data record for the square footage of a house 18,000 square feet and its valuation \$260,000, we could express that as a vector [18000 2600000], stepping 18,000 steps in the horizontal direction and 260,000 steps in the vertical direction.

We declare a vector mathematically like this:

$$\vec{v} = \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\vec{v} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

We can declare a vector using a simple Python collection, like a Python list as shown in [Example 4-1](#).

Example 4-1. Declaring a vector in Python using a list

```
v = [3, 2]
print(v)
```

However when we start doing mathematical computations with vectors, especially when doing tasks like machine learning, we should probably use the NumPy library as it is more efficient than plain Python. You can also use SymPy to perform linear algebra operations, and we will use it occasionally in this chapter when decimals become inconvenient. However, NumPy is what you will likely use in practice so that is what we will mainly stick to.

To declare a vector, you can use NumPy's `array()` function and then can pass a collection of numbers to it as shown in [Example 4-2](#).

Example 4-2. Declaring a vector in Python using NumPy

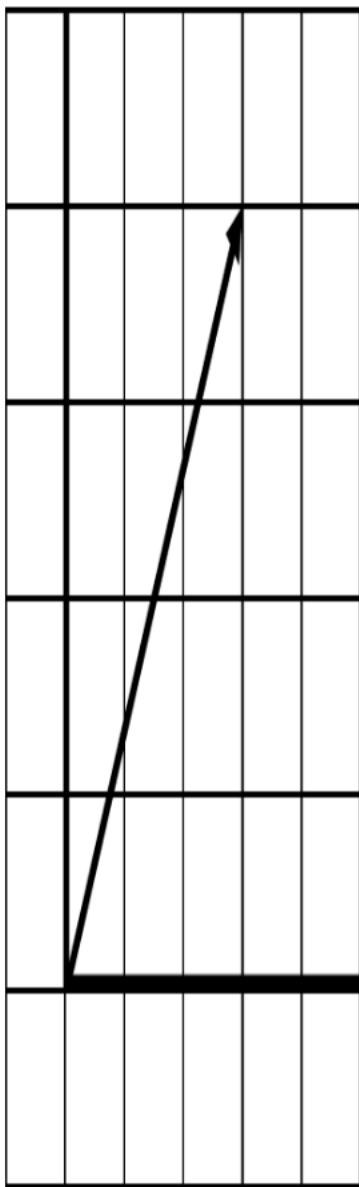
```
import numpy as np
v = np.array([3, 2])
print(v)
```

PYTHON LIBRARIES ARE MORE EFFICIENT

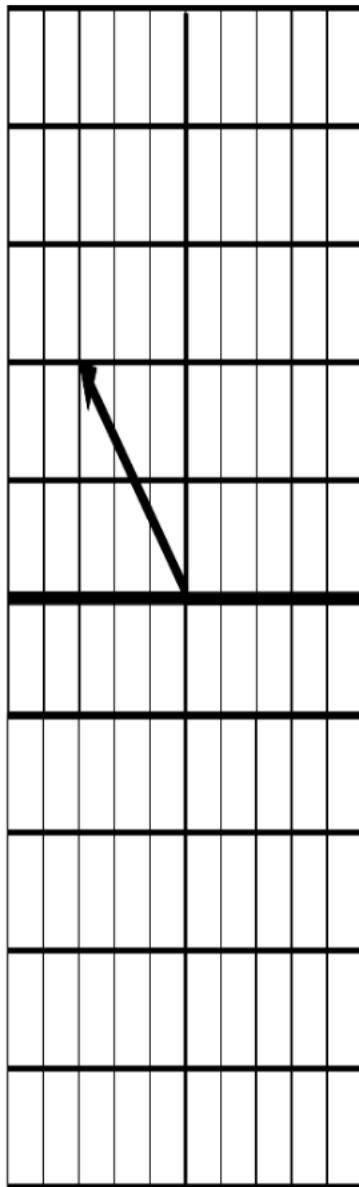
Python is slow but its numeric and scientific libraries are not. Libraries like NumPy are typically written in low-level languages like C and C++, hence why they are computationally efficient. Python really acts as “glue code” integrating these libraries for your task.

A vector has countless practical applications. In physics, a vector is often thought of as a direction and magnitude. In math it is a direction and scale on an XY plane, kind of like a movement. In computer science, it is an array of numbers storing data. The computer science context is the one we will become the most familiar with as data science professionals. However it is important we never forget the visual aspect so we do not think of vectors as esoteric grids of numbers. Without a visual understanding it is almost impossible to grasp many fundamental linear algebra concepts like linear dependence and determinants.

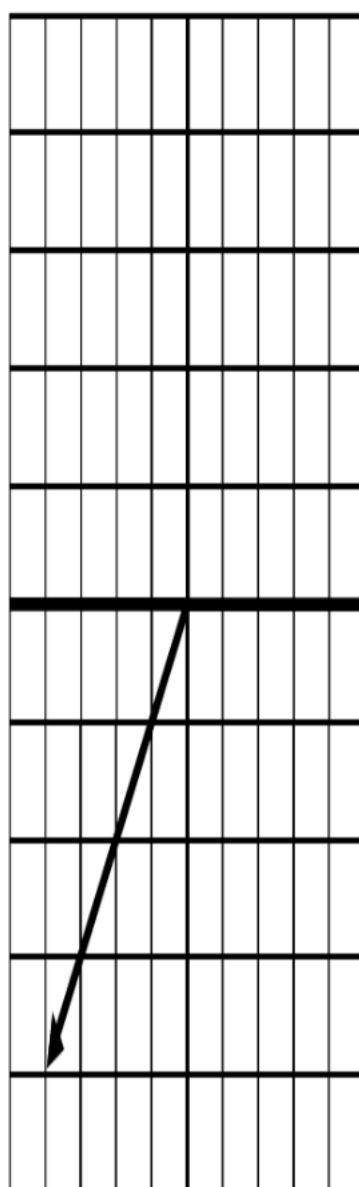
Here are some more examples of vectors. In [Figure 4-2](#) note that some of these vectors have negative directions on the X and Y scales. Vectors with negative directions will have an impact when we combine them later, essentially subtracting rather than adding them together.



$$\vec{v} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$



$$\vec{v} = \begin{bmatrix} -3 \\ 2 \end{bmatrix}$$



$$\vec{v} = \begin{bmatrix} -4 \\ 4 \end{bmatrix}$$

Figure 4-2. A sampling of different vectors

WHY ARE VECTORS USEFUL?

A struggle many people have with vectors (and linear algebra in general) is understanding why it is useful. It's a highly abstract concept but it has many tangible applications. Computer graphics are easier to work with when you have a grasp of vectors and linear transformations. When doing statistics and machine learning work, data is often imported and turned into numerical vectors so we can work with it. Solvers like the one in Excel or Python PuLP use linear programming, which uses vectors to maximize a solution while meeting those constraints. Even video games and flight simulators use vectors and linear algebra not just to model graphics but also physics. I think what makes vectors so hard to grasp is not that their application is unobvious, but rather these applications are so diverse it is hard to see the generalization.

Note also vectors can exist on more than two dimensions. Below we declare a 3-dimensional vector along axes x, y, and z:

$$\vec{v} = \begin{vmatrix} x \\ y \\ z \end{vmatrix} = \begin{vmatrix} 4 \\ 1 \\ 2 \end{vmatrix}$$

Note to create the vector above we are stepping 4 steps in the x direction, 1 in the y direction, and 2 in the z direction. Here it is visualized in [Figure 4-3](#). Note that we no longer are showing a vector on a 2-dimensional grid but rather a 3-dimensional space with 3 axes: x, y, and z.

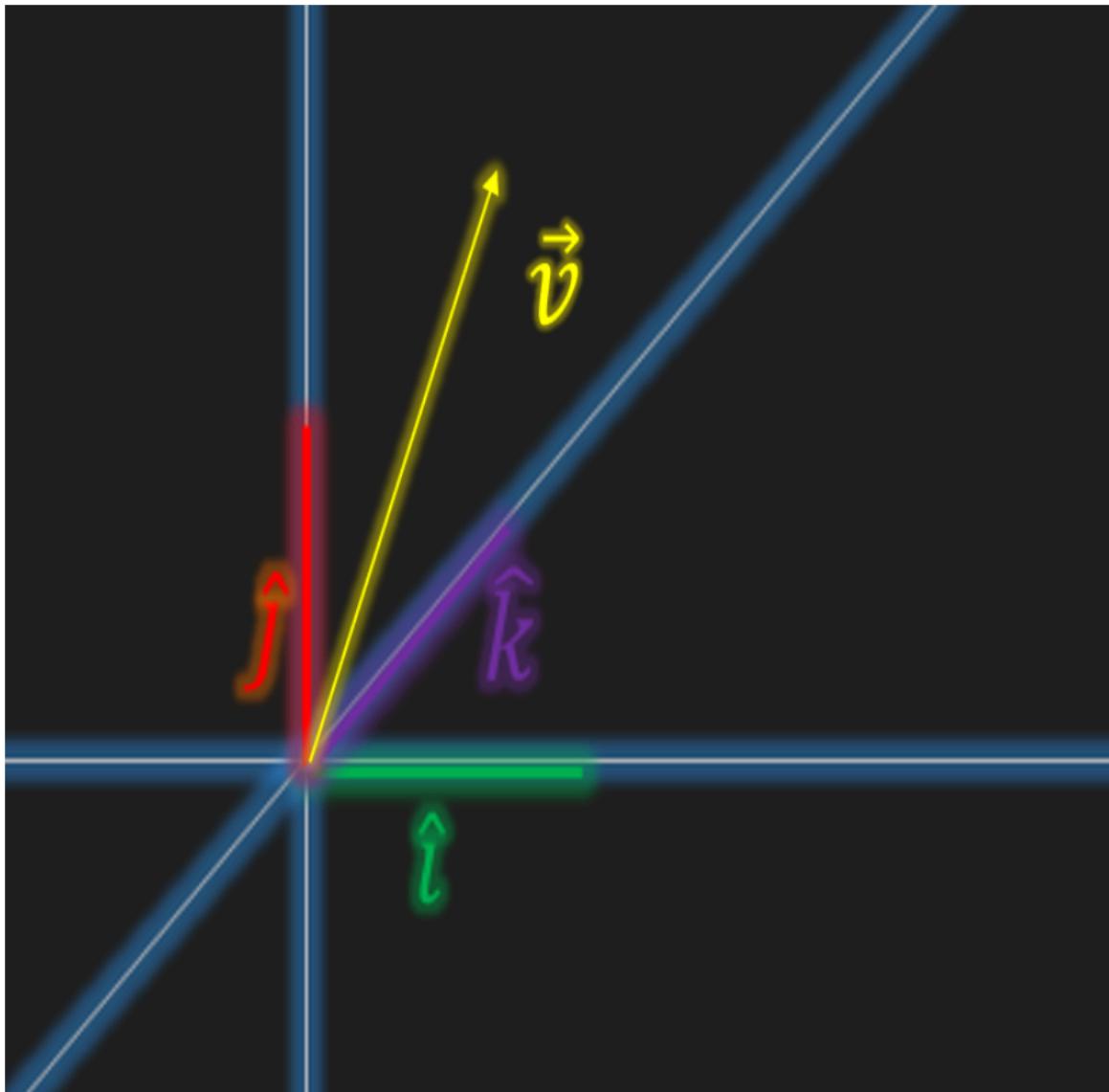


Figure 4-3. A 3-dimensional vector

Naturally we can express this 3-dimensional vector in Python using 3 numeric values, as declared in [Example 4-3](#).

[Example 4-3. Declaring a 3-dimensional vector in Python using NumPy](#)

```
import numpy as np  
v = np.array([4, 1, 2])  
print(v)
```

Like many mathematical models, visualizing more than 3 dimensions is challenging and something we will not expense energy doing so in this book.

But numerically it is still straightforward. [Example 4-4](#) shows how we declare a 5-dimensional vector mathematically in Python.

$$\vec{v} = \begin{bmatrix} 6 \\ 1 \\ 5 \\ 8 \\ 3 \end{bmatrix}$$

[Example 4-4. Declaring a 5-dimensional vector in Python using NumPy](#)

```
import numpy as np
v = np.array([6, 1, 5, 8, 3])
print(v)
```

Adding and Combining Vectors

On their own, vectors are not terribly interesting. You express a direction and size, kind of like a movement of space. But when you start combining vectors, known as **vector addition**, it starts to get interesting. We effectively combine the movements of two vectors into a single vector.

Say we have two vectors \vec{v} and \vec{w} as shown in [Figure 4-4](#). How do we add these two vectors together?

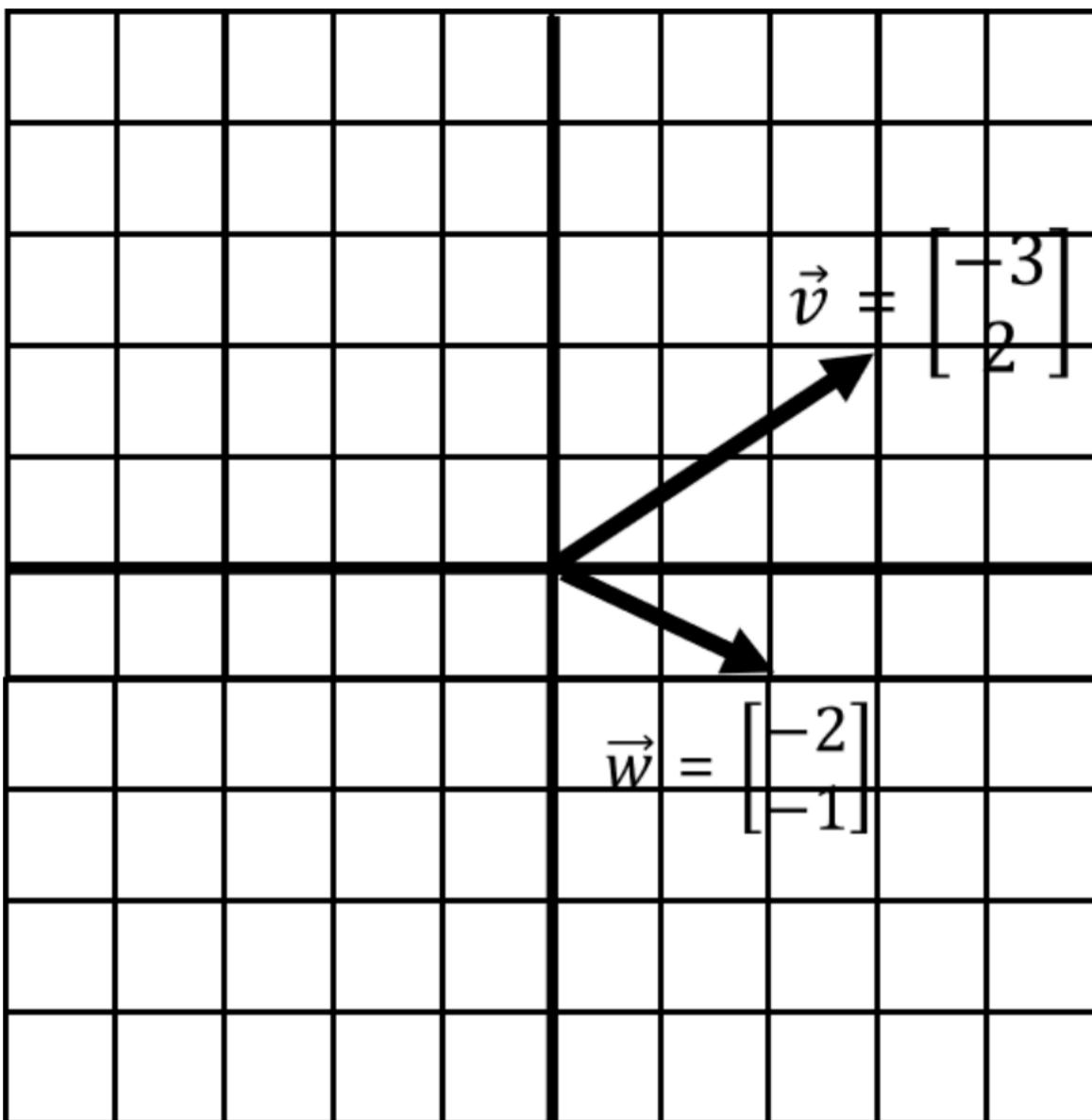


Figure 4-4. Adding two vectors together

We will get to why adding vectors is useful in a moment. But if we wanted to combine these two vectors, including their direction and scale, what would that look like? Numerically this is straightforward. You simply add the respective x values and then the y values into a new vector as shown in [Example 4-5](#).

$$\vec{v} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

$$\vec{w} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$$

$$\vec{v} + \vec{w} = \begin{bmatrix} 3 + 2 \\ 2 + -1 \end{bmatrix} = \begin{bmatrix} 5 \\ 1 \end{bmatrix}$$

Example 4-5. Adding two vectors in Python using NumPy

```
from numpy import array
```

```
v = array([3,2])  
w = array([2,-1])
```

```
# sum the vectors  
v_plus_w = v + w
```

```
# display summed vector  
print(v_plus_w) # [5, 1]
```

But what does this mean visually? To visually add these two vectors together, connect one vector after the other and walk to the tip of the last vector (**Figure 4-5**). The point you end at is a new vector, the result of summing the two vectors.

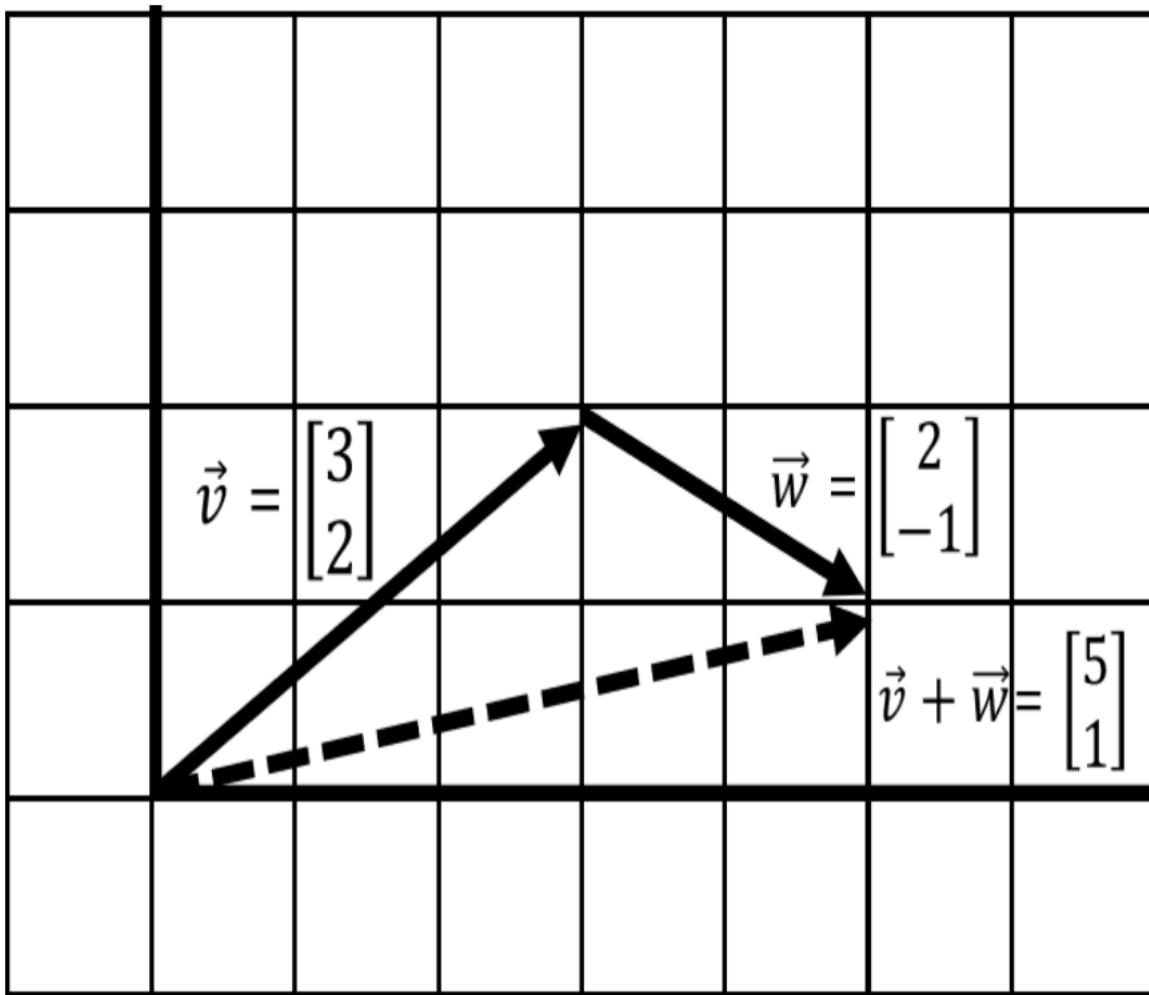


Figure 4-5. Adding two vectors into a new vector

As seen above in [Figure 4-5](#), when we walk to the end of the last vector \vec{w} we end up with a new vector $[5, 1]$. This new vector is the result of summing \vec{v} and \vec{w} . In practice, this can be simply adding data together. If we were totaling housing values and their square footage in an area, we would be adding several vectors into a single vector in this manner.

Note that it does not matter whether we add \vec{v} before \vec{w} or vice versa, which means it is **commutative** and order of operation does not matter. If we walk \vec{w} before \vec{v} we end up with the same resulting vector $[5, 1]$ as visualized in [Figure 4-6](#).

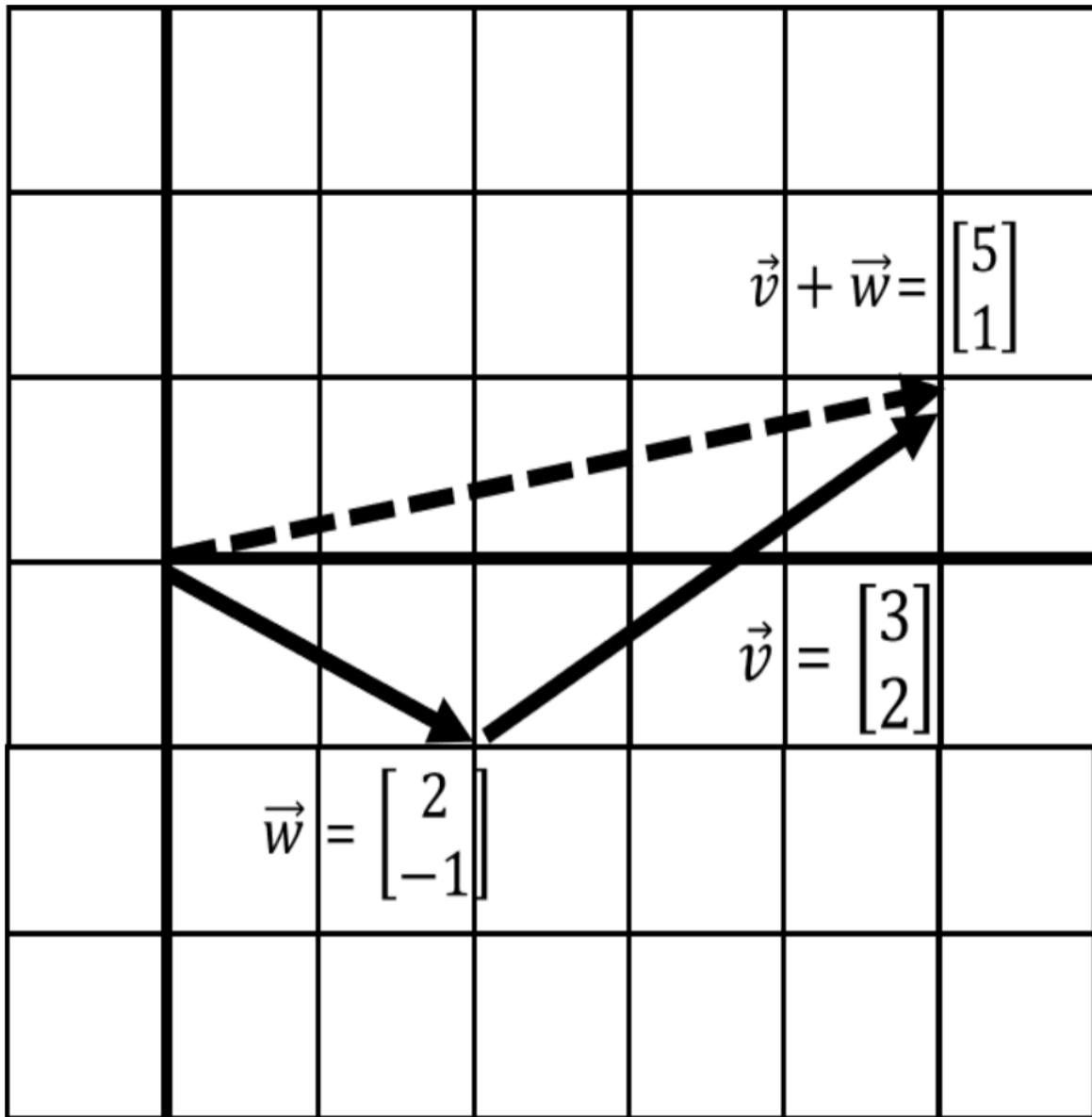


Figure 4-6. Adding vectors is commutative

Scaling Vectors

Scaling is growing or shrinking a vector's length. You can grow/shrink a vector by multiplying or scaling it with a single value, known as a **scalar**.

Figure 4-7 is vector \vec{v} being scaled by a factor of 2, which doubles it.

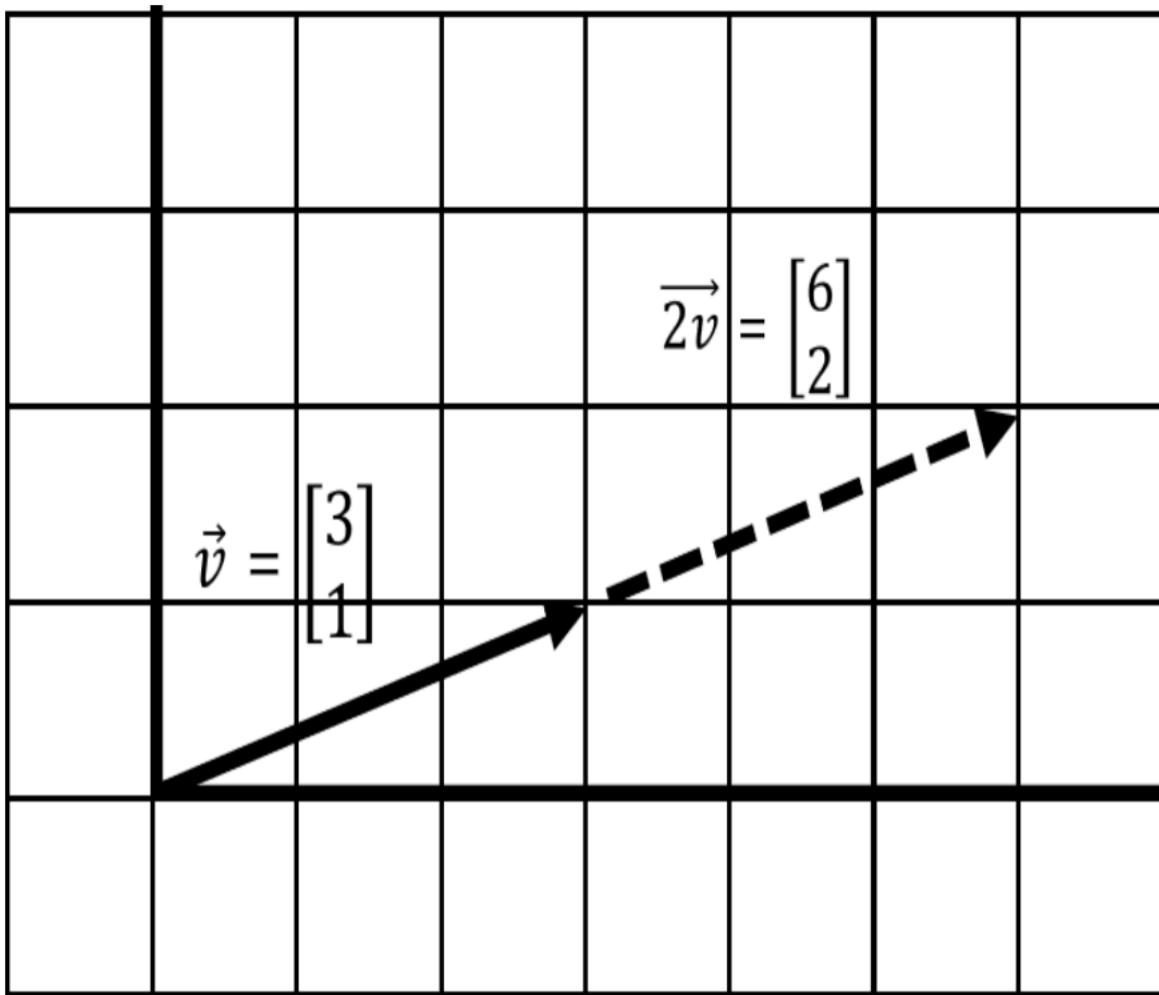


Figure 4-7. Scaling a vector

Mathematically you multiply each element of the vector by the scalar value.

$$\vec{v} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

$$2\vec{v} = 2 \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 \times 2 \\ 2 \times 2 \end{bmatrix} = \begin{bmatrix} 6 \\ 4 \end{bmatrix}$$

Performing this scaling operation in Python is as easy as multiplying a vector by the scalar, as coded in [Example 4-6](#).

[Example 4-6. Scaling a number in Python using NumPy](#)

```
from numpy import array
```

```
v = array([3,1])  
  
# scale the vector  
scaled_v = 2.0 * v  
  
# display scaled vector  
print(scaled_v) # [6 2]
```

Here in [Figure 4-8](#) is \vec{v} being scaled down by factor of .5, which halves it.

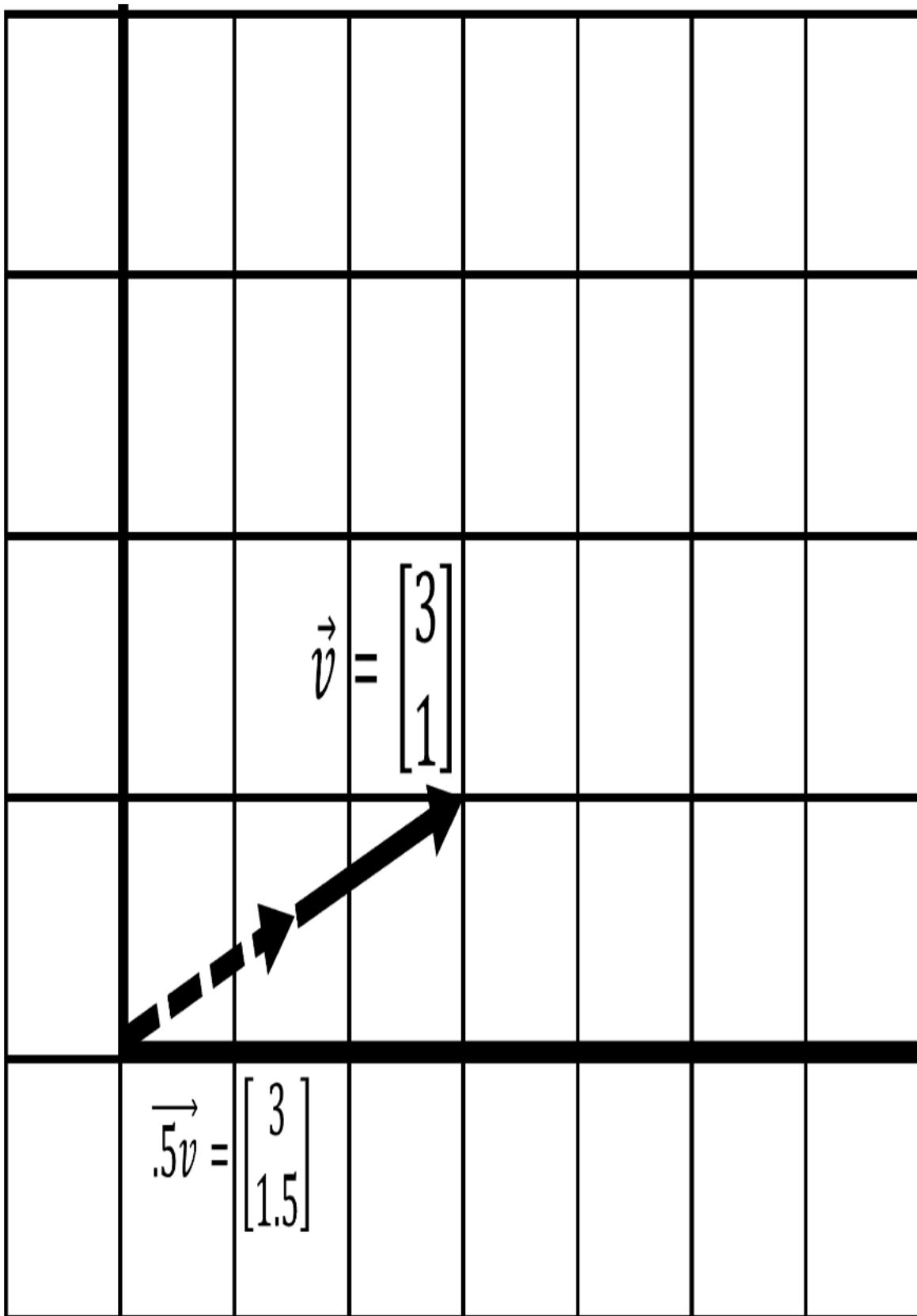


Figure 4-8. Scaling down a vector by half

MANIPULATING DATA IS MANIPULATING VECTORS

Every data operation can be thought of in terms of vectors, even simple averages.

Take scaling for example. Let's say we were trying to get the average house value and average square footage for an entire neighborhood. We would add the vectors together to combine their value and square footage respectively, giving us one giant vector containing both total value and total square footage. We then scale down the vector by dividing by the number of houses N , which really is multiplying by $\frac{1}{N}$. We now have a vector containing the average house value and average square footage.

An important detail to note here is that scaling a vector does not change its direction, only its magnitude. But there is one slight exception to this rule as visualized in [Figure 4-9](#). When you multiply a vector by a negative number, it flips the direction of the vector as shown below:

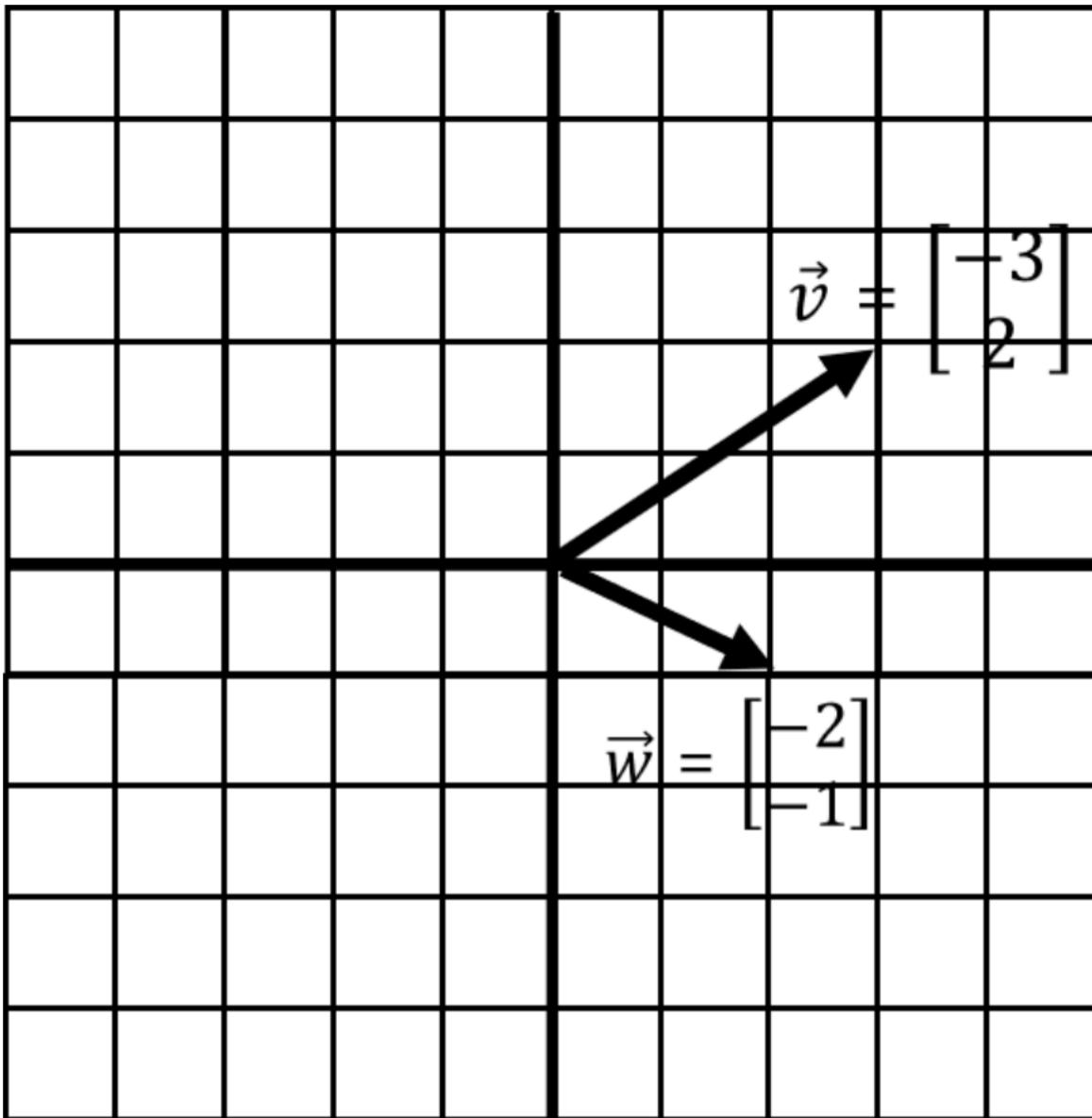


Figure 4-9. A negative scalar flips the vector direction

When you think about it though, scaling by a negative number has not really changed direction in that it still exists on the same line. This segues us into a key concept called linear dependence.

Span and Linear Dependence

These two operations, adding two vectors and scaling them, brings about a simple but powerful idea. With these two operations, we can combine two vectors and scale them to create any resulting vector we want. [Figure 4-10](#)

shows six examples of taking two vectors \vec{v} and \vec{w} , and scaling and combining. These vectors \vec{v} and \vec{w} , fixed in two different directions, can be scaled and added to create ANY new vector $\overrightarrow{v+w}$.

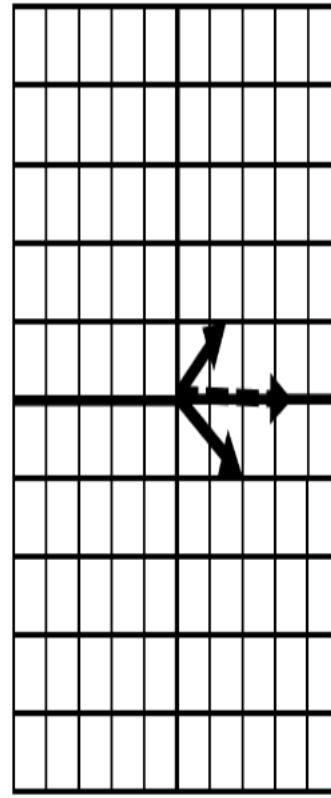
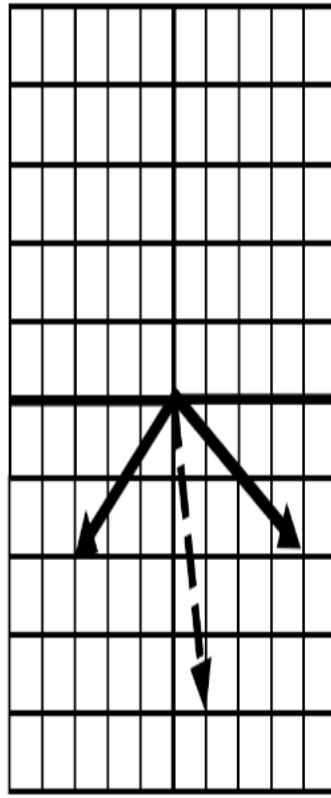
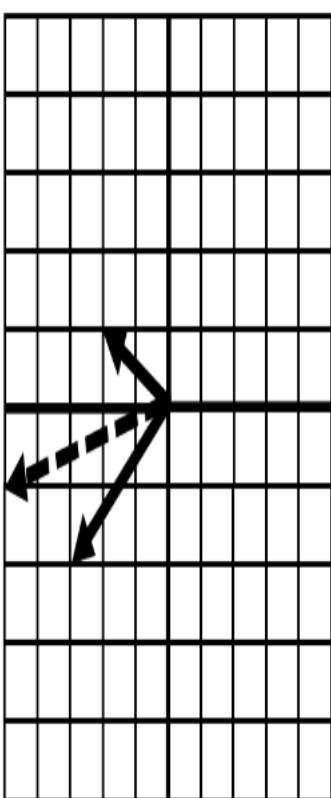
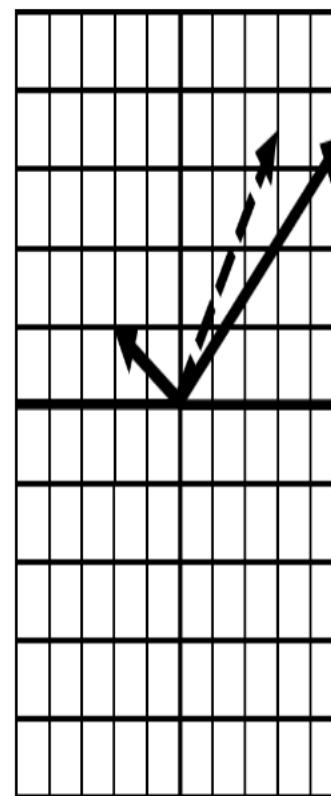
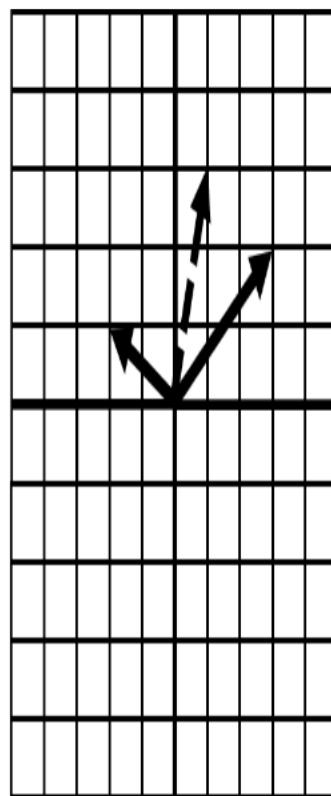
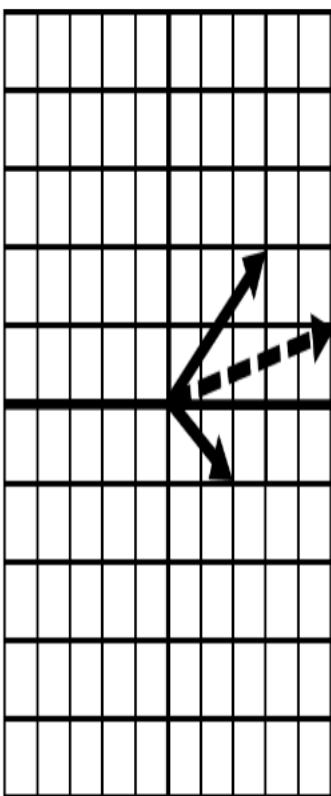


Figure 4-10. Scaling two added vectors allows us to create any new vector

AGAIN, \vec{v} and \vec{w} are fixed in direction, except for flipping with negative scalars, but we can use scaling to freely create any vector composed of $\overrightarrow{v + w}$.

This whole space of possible vectors is called **span**, and in most cases our span can create unlimited vectors off those two vectors, simply by scaling and summing them. When we have two vectors in two different directions, they are **linearly independent** and have this unlimited span.

But in what case are we limited in the vectors we can create? Think about it and read on.

What happens when two vectors exist in the same direction, or exist on the same line? The combination of those vectors is also stuck on the same line, limiting our span to just that line. No matterhow you scale it, the resulting sum vector is also stuck on that same line. This makes them **linearly dependent** as shown in [Figure 4-11](#).

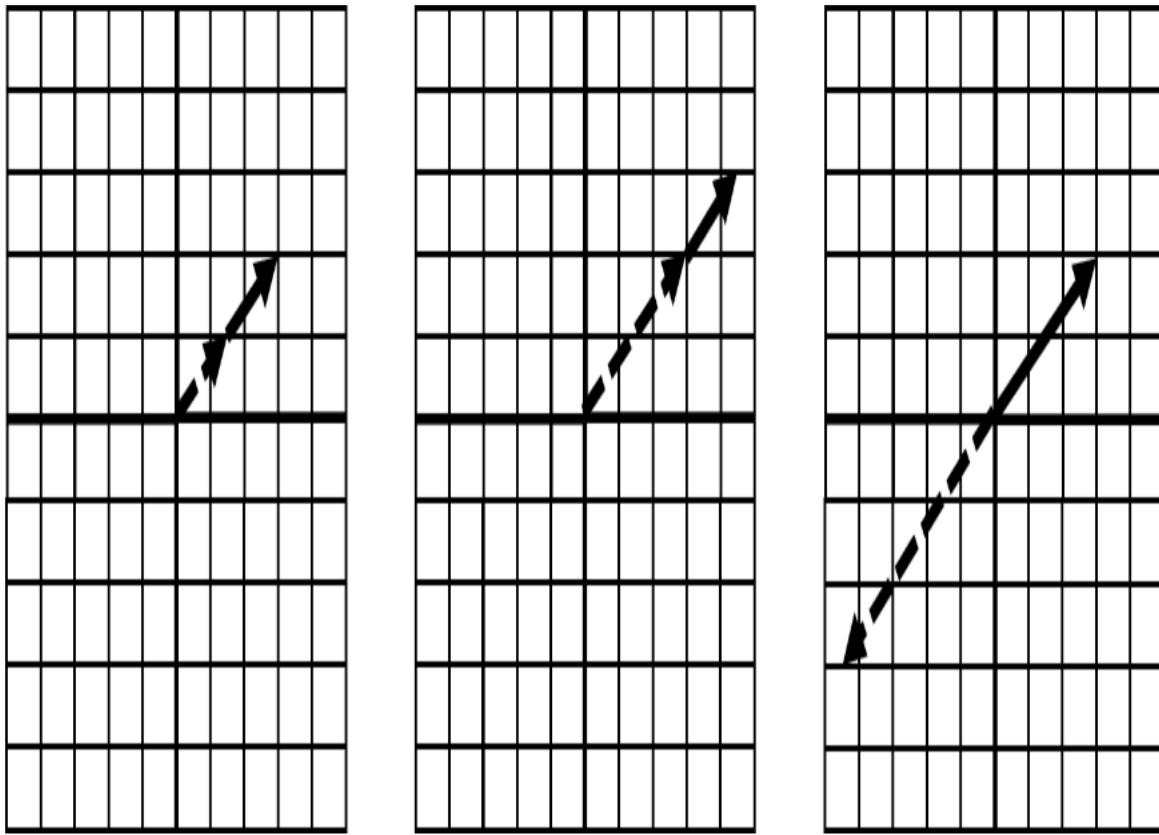


Figure 4-11. Linearly dependent vectors

The span here is stuck on the same line as the two vectors it is made out of. Because the two vectors exist in the same direction (or share the same line), we cannot flexibly create any new vector through scaling. Any resulting vector is stuck on that line.

In 3 or more dimensions, when we have a linearly dependent set of vectors we often get stuck on a plane in a smaller number of dimensions. Here is an example of being stuck on a 2-dimensional plane even though we have 3-dimensional vectors as declared in [Figure 4-12](#).

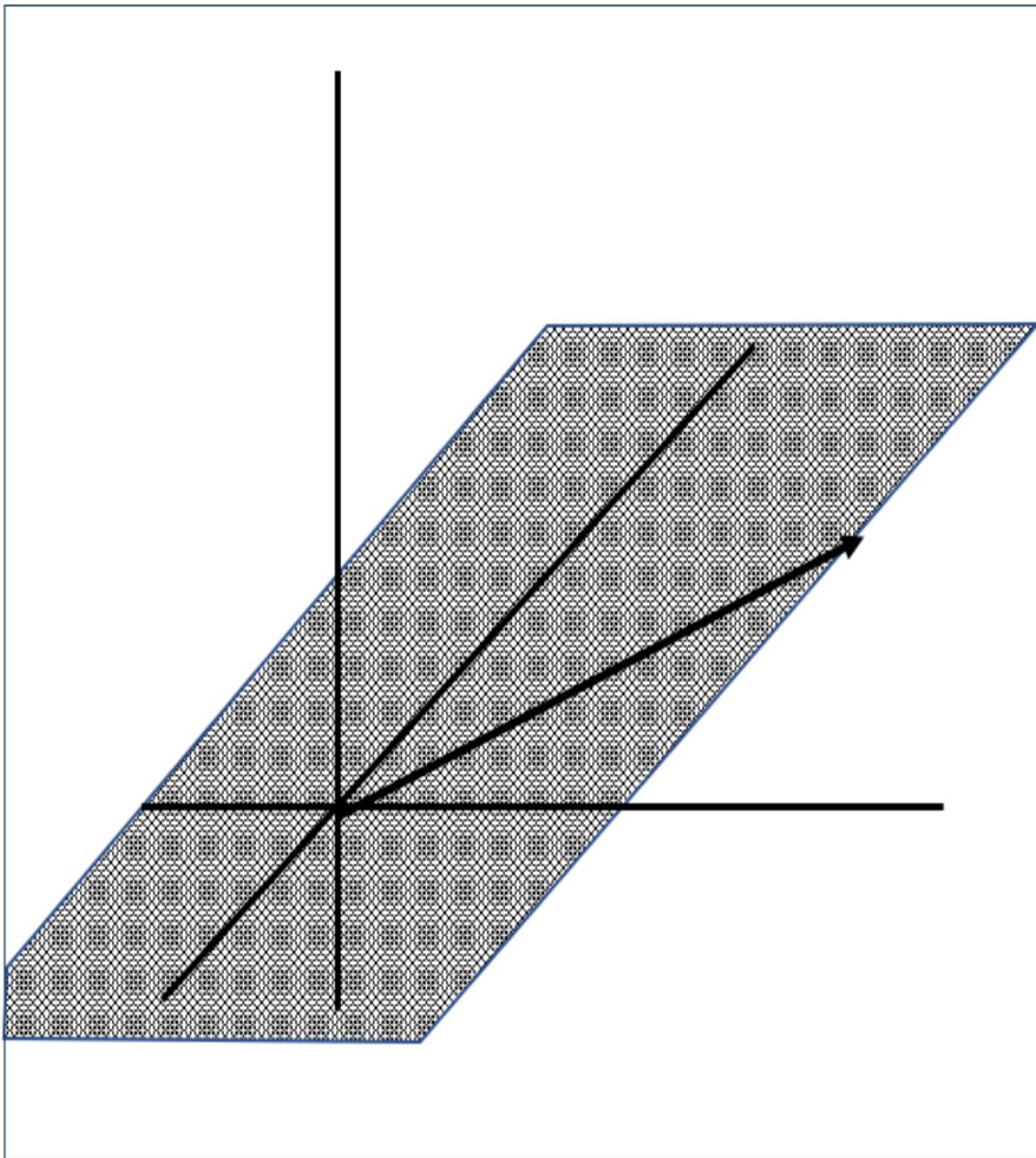


Figure 4-12. Linear dependence in 3 dimensions, note our span is limited to a flat plane

Later we will learn a simple tool called the determinant to check for linear dependence, but why do we care whether two vectors are linearly dependent or independent? A lot of problems become difficult or unsolvable when they are linearly dependent. For example, when we learn about systems of equations later in this chapter, a linearly dependent set of equations can cause variables to disappear and make the problem unsolvable. But if you have

linear independence, that flexibility to create any vector you need from two or more vectors becomes invaluable to solve for a solution!

Linear Transformations

This concept of adding two vectors with fixed direction, but scaling them to get different combined vectors, is hugely important. This combined vector, except in cases of linear dependence, can point in any direction and have any length we choose. This sets up an intuition for linear transformations where we use a vector to transform another vector in a functional-like manner.

Basis Vectors

Imagine we have two simple vectors \hat{i} and \hat{j} (“i-hat” and “j-hat”). These are known as **basis vectors**, which are used to describe transformations on other vectors. They typically have a length of 1 and point in perpendicular positive directions as visualized in [Figure 4-13](#).

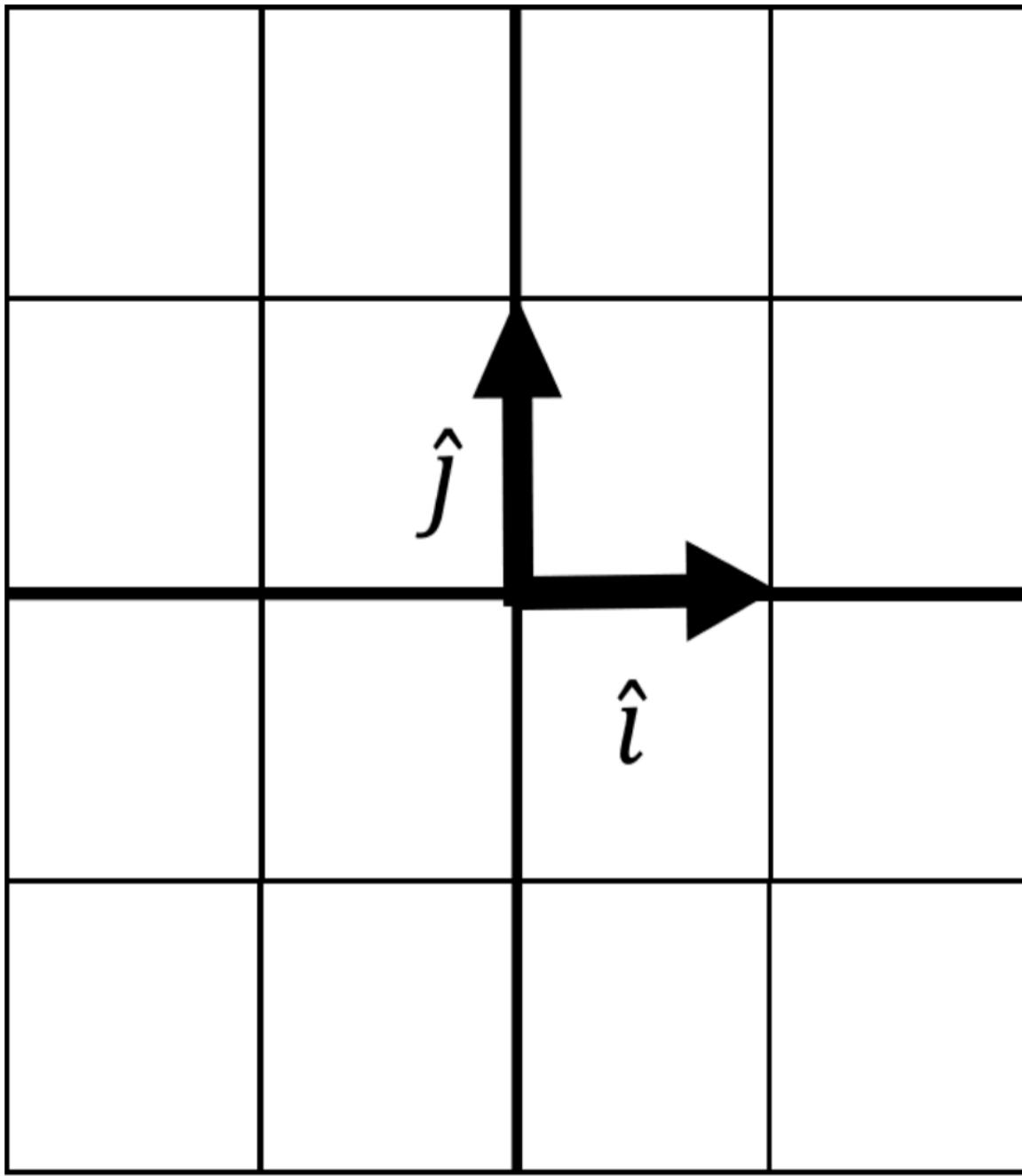


Figure 4-13. Basis vectors \hat{i} and \hat{j}

Think of the basis vectors as building blocks to build or transform any vector. Our basis vector is expressed in a 2×2 matrix, where the first column is \hat{i} and the second column is \hat{j} .

$$\hat{i} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\hat{j} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\text{basis} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

A **matrix** is a collection of vectors (such as \hat{i} , \hat{j}), can have multiple rows and columns, and is a convenient way to package data. We can use \hat{i} and \hat{j} to create any vector we want by scaling and adding them. Let's start with each having a length of 1 and showing the resulting vector \vec{v} in [Figure 4-14](#).

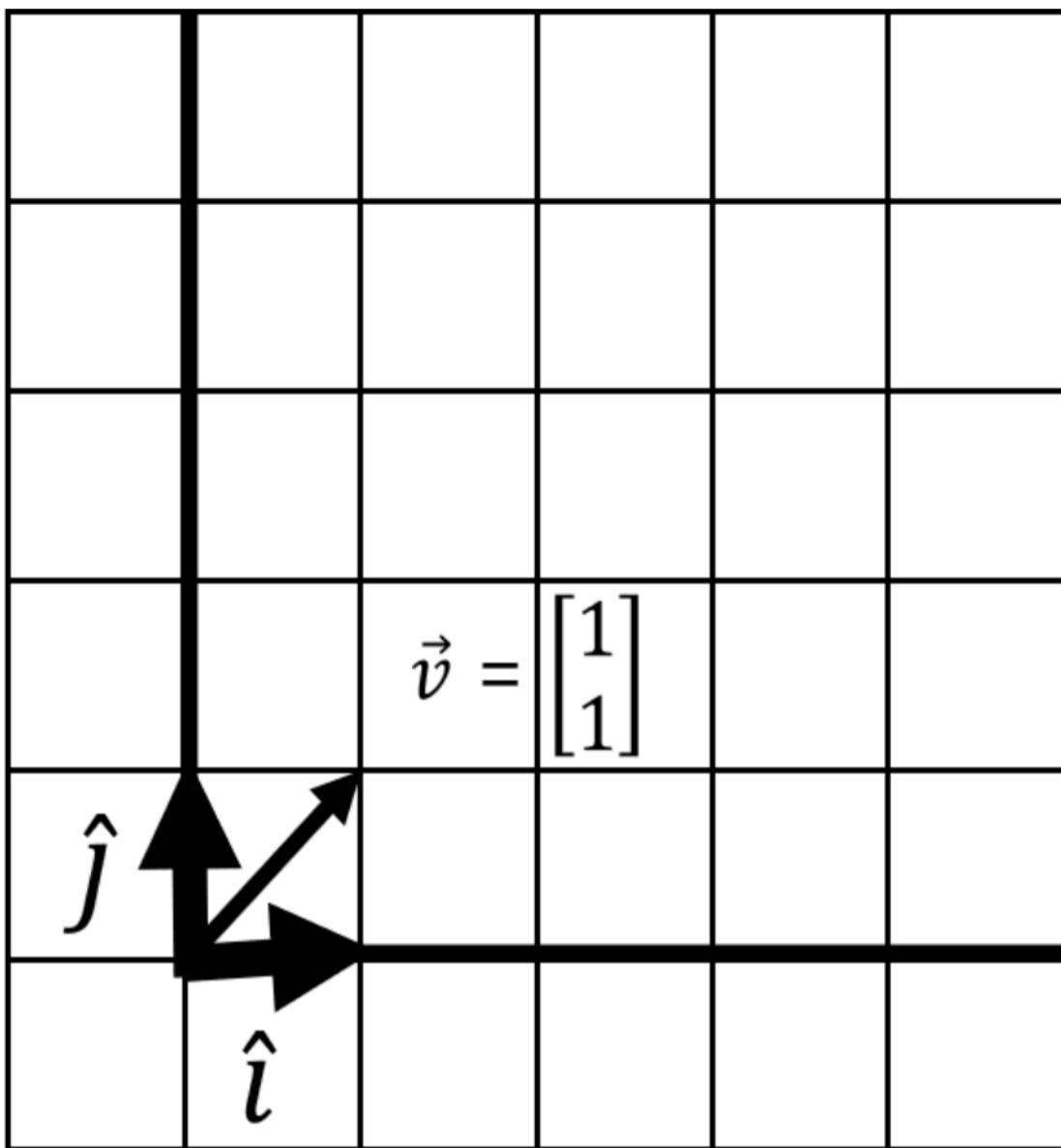


Figure 4-14. Creating a vector off of basis vectors

I want vector \vec{v} to land at [3, 2]. What happens to \vec{v} if we stretch \hat{i} by a factor of 3 and \hat{j} by a factor of 2? First we scale them individually as shown here:

$$3\hat{i} = 3 \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 0 \end{bmatrix}$$

$$2\hat{j} = 2 \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$$

If we stretched space in these two directions, what does this do to \vec{v} ? Well it is going to stretch with \hat{i} and \hat{j} . This is known as a **linear transformation**, where we transform a vector with stretching, squishing, sheering, or rotating by tracking basis vector movements. In this case of [Figure 4-15](#), scaling \hat{i} and \hat{j} has stretched space along with our vector \vec{v} .

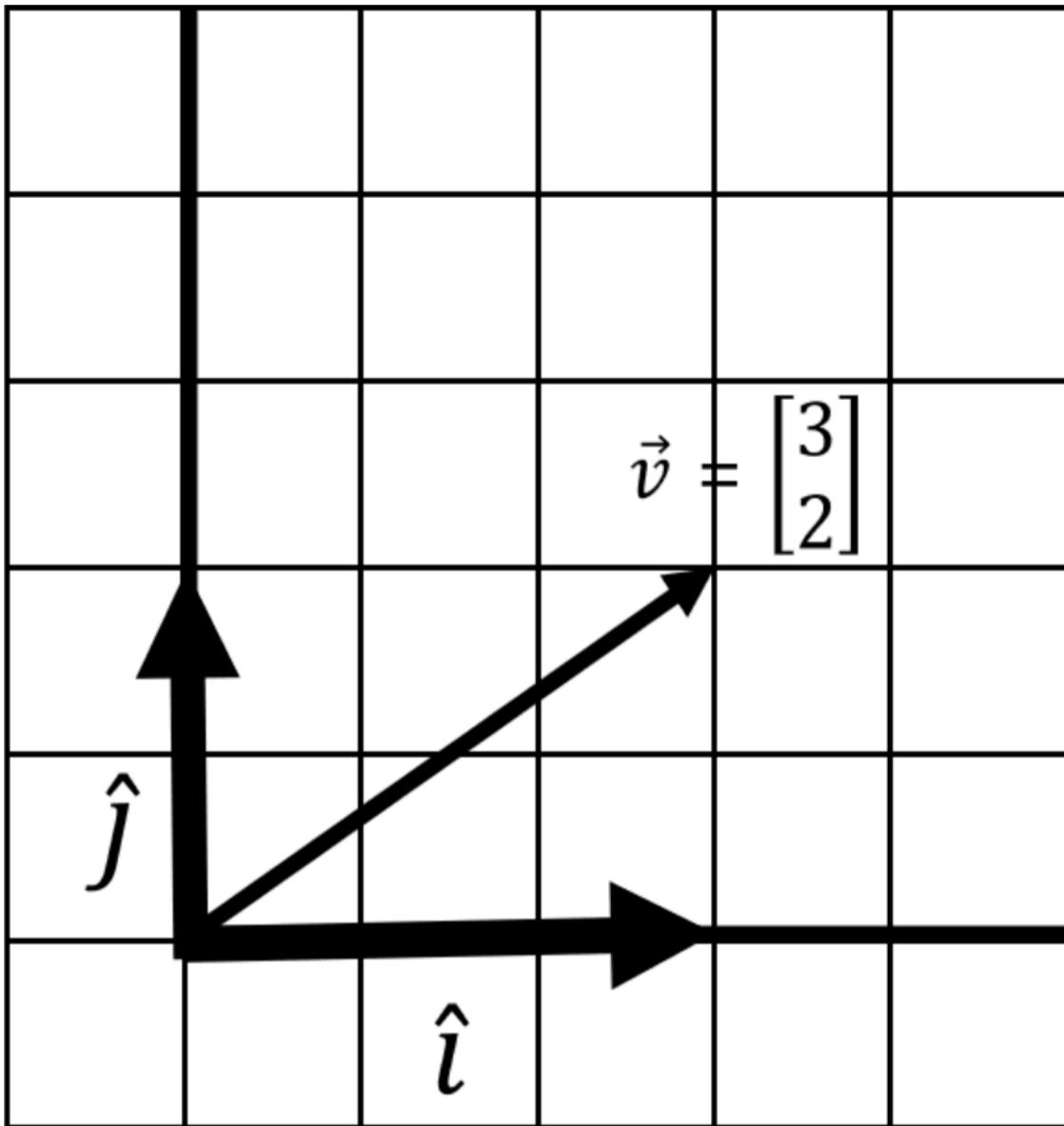
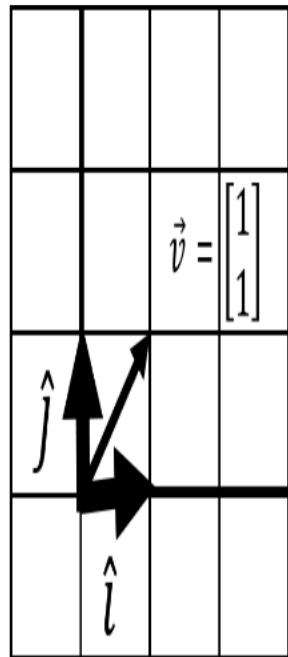


Figure 4-15. A linear transformation

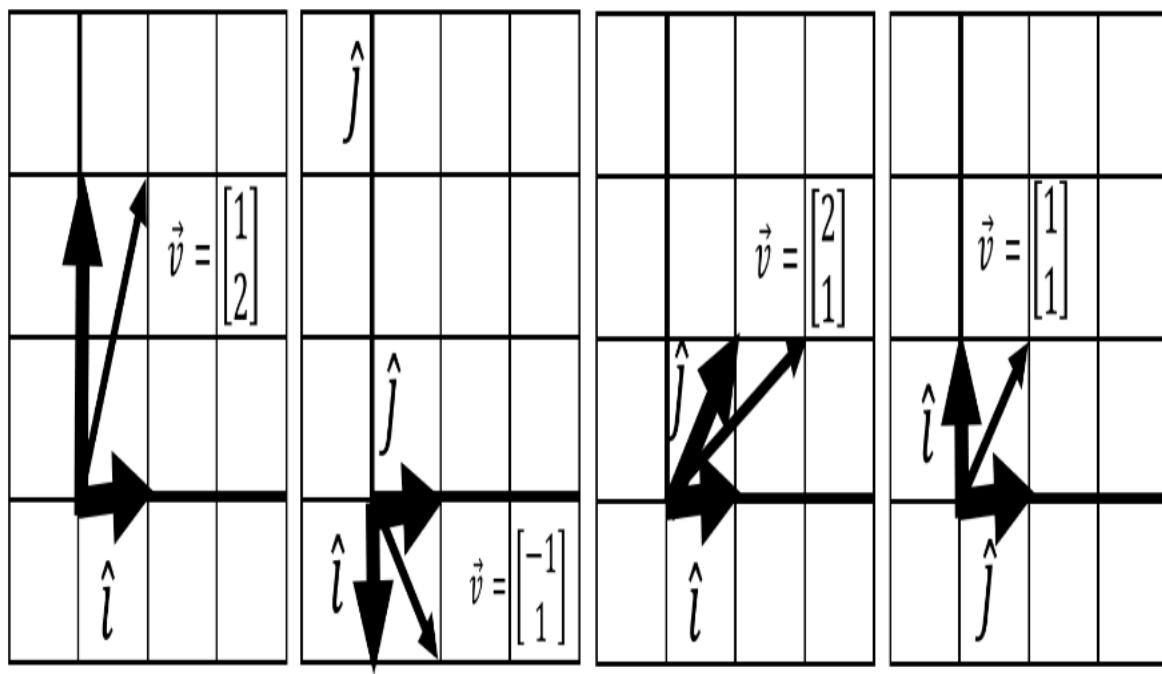
But where does \vec{v} land? It is easy to visually see where it lands here, which is $[3, 2]$. Recall that vector \vec{v} is composed off of adding \hat{i} and \hat{j} . So we simply take the stretched \hat{i} and \hat{j} and add them together to see where vector \vec{v} has landed.

$$\vec{v}_{new} = \begin{bmatrix} 3 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

Generally with linear transformations, there are four movements you can achieve shown in [Figure 4-16](#).



BASIS



SCALE

ROTATE

SHEAR

INVERSION

Figure 4-16. Four movements can be achieved with linear transformations

These four linear transformations are a central part of linear algebra. Scaling a vector will stretch or squeeze it. Rotations will turn the vector space, and inversions will flip the vector space so that \hat{i} and \hat{j} swap respective places.

It is important to note that you cannot have transformations that are nonlinear, resulting in curvy or squiggly transformations that no longer respect a straight line. This is why we call it linear algebra, not nonlinear algebra!

Matrix Vector Multiplication

This brings us to our next big idea in linear algebra. This concept of tracking where \hat{i} and \hat{j} land after a transformation is important because it allows us not just to create vectors but also transform existing vectors. If you want true linear algebra enlightenment, think why creating vectors and transforming vectors are actually the same thing. It's all a matter of relativity given your basis vectors being a starting point before and after a transformation.

The formula to transform a vector \vec{v} given basis vectors \hat{i} and \hat{j} packaged as a matrix is as follows:

$$\begin{bmatrix} x_{new} \\ y_{new} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} x_{new} \\ y_{new} \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

\hat{i} is the first column $[a \ c]$ and \hat{j} is the column $[b \ d]$. We package both of these basis vectors as a matrix, which again is a collection of vectors expressed as a grid of numbers in two or more dimensions. This transformation of a vector by applying basis vectors is known as **matrix vector multiplication**. This may seem contrived at first, but this formula is a shortcut for scaling and adding \hat{i} and \hat{j} just like we did earlier adding two vectors, and applying the transformation to any vector \vec{v} . \hat{i} is the first column

$[a \ c]$ and \hat{j} is the column $[b \ d]$. We package both of these basis vectors as a matrix, which again is a collection of vectors expressed as a grid of numbers in two or more dimensions. This transformation of a vector by applying basis vectors is known as **matrix vector multiplication**. This may seem contrived at first, but this formula is a shortcut for scaling and adding \hat{i} and \hat{j} just like we did earlier adding two vectors, and applying the transformation to any vector \vec{v} . So in effect, a matrix really is a transformation expressed as basis vectors.

To execute this transformation in Python using NumPy, we will need to declare our basis vectors as a matrix and then apply it to vector \vec{v} using the `dot()` operator ([Example 4-7](#)). The dot operator will perform this scaling and addition between our matrix and vector as we just described above. This is known as the **dot product** and we will explore it throughout this chapter.

Example 4-7. Matrix vector multiplication in NumPy

```
from numpy import array

# compose basis matrix with i-hat and j-hat
basis = array(
    [[3, 0],
     [0, 2]]
)

# declare vector v
v = array([1,1])

# create new vector
# by transforming v with dot product
new_v = basis.dot(v)

print(new_v) # [3, 2]
```

When thinking in terms of basis vectors, I prefer to break out the basis vectors and then compose them together into a matrix. Just note you will need to **transpose**, or swap the columns and rows. This is because NumPy's `array()` function will do the opposite orientation we want, populating each vector as a row rather than a column. Transposition in NumPy is demonstrated in [Example 4-8](#).

Example 4-8.

```
from numpy import array

# Declare i-hat and j-hat
i_hat = array([2, 0])
j_hat = array([0, 3])

# compose basis matrix using i-hat and j-hat
# also need to transpose rows into columns
basis = array([i_hat, j_hat]).transpose()

# declare vector v
v = array([1,1])

# create new vector
# by transforming v with dot product
new_v = basis.dot(v)

print(new_v) # [2, 3]
```

Here's another example. Let's start with vector \vec{v} being $[2 \ 1]$ and \hat{i} and \hat{j} start at $[1 \ 0]$ and $[0 \ 1]$ respectively. We then transform \hat{i} and \hat{j} to $[2 \ 0]$ and $[0 \ 3]$. What happens to vector \vec{v} ? Working this out mathematically by hand using our formula, we get this.

$$\begin{bmatrix} x_{new} \\ y_{new} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

$$\begin{bmatrix} x_{new} \\ y_{new} \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} (2)(2) + (0)(1) \\ (2)(0) + (3)(1) \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \end{bmatrix}$$

Example 4-9 shows this solution in Python.

Example 4-9. Transforming a vector using NumPy

```
from numpy import array

# Declare i-hat and j-hat
i_hat = array([2, 0])
j_hat = array([0, 3])

# compose basis matrix using i-hat and j-hat
# also need to transpose rows into columns
```

```
basis = array([i_hat, j_hat]).transpose()

# declare vector v
v = array([2,1])

# create new vector
# by transforming v with dot product
new_v = basis.dot(v)

print(new_v) # [4 3]
```

The vector \vec{v} now lands at $[4 \ 3]$. Figure 4-17 shows a visual of what this transformation looks like.

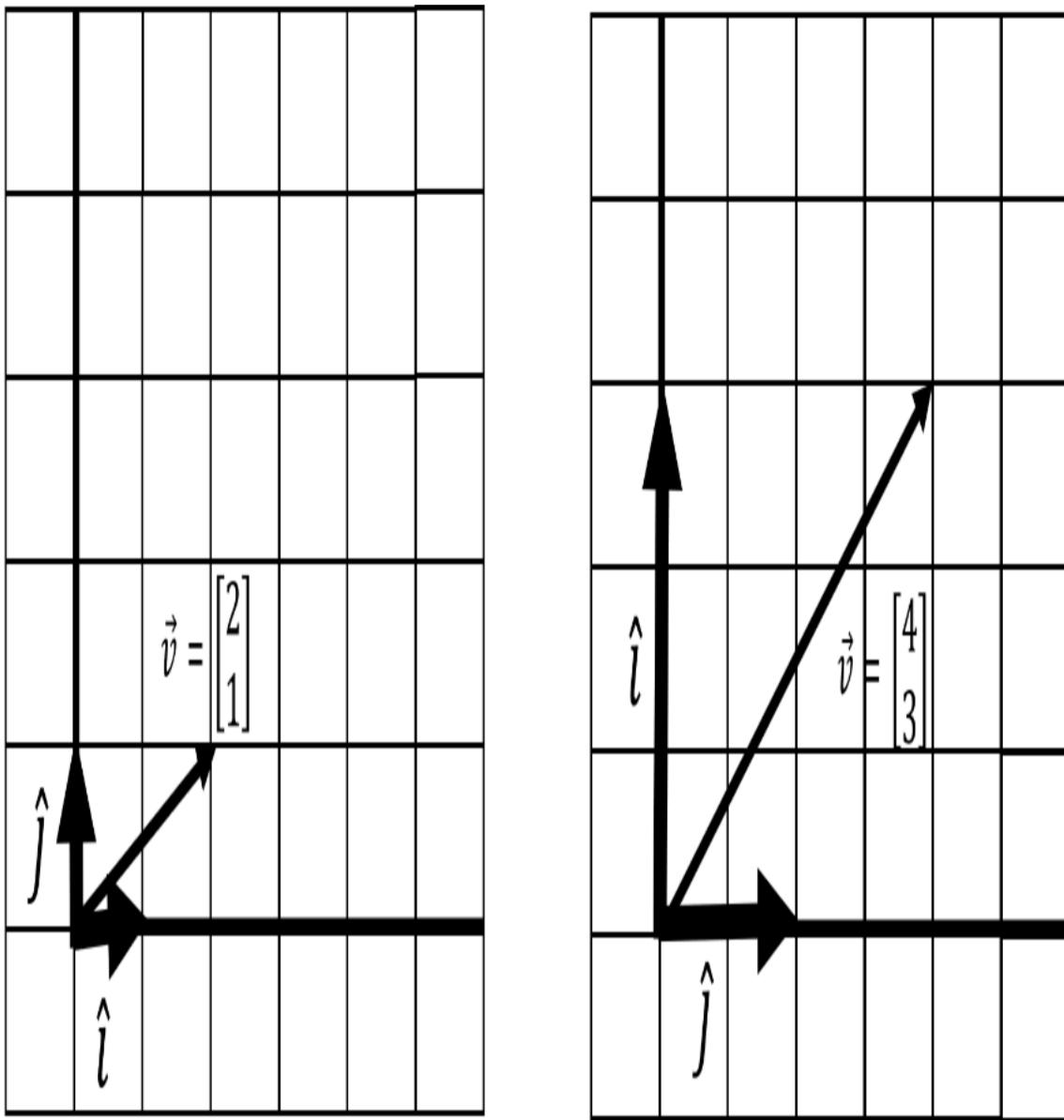


Figure 4-17. A stretching linear transformation

Here is an example that jumps things up a notch. Let's take vector \vec{v} of value $[2 \ 1]$. \hat{i} and \hat{j} start at $[1 \ 0]$ and $[0 \ 1]$, but then are transformed and land at $[2 \ 3]$ and $[2 \ -1]$. What happens to \vec{v} ? Let's look in [Figure 4-18](#) and [Example 4-10](#).

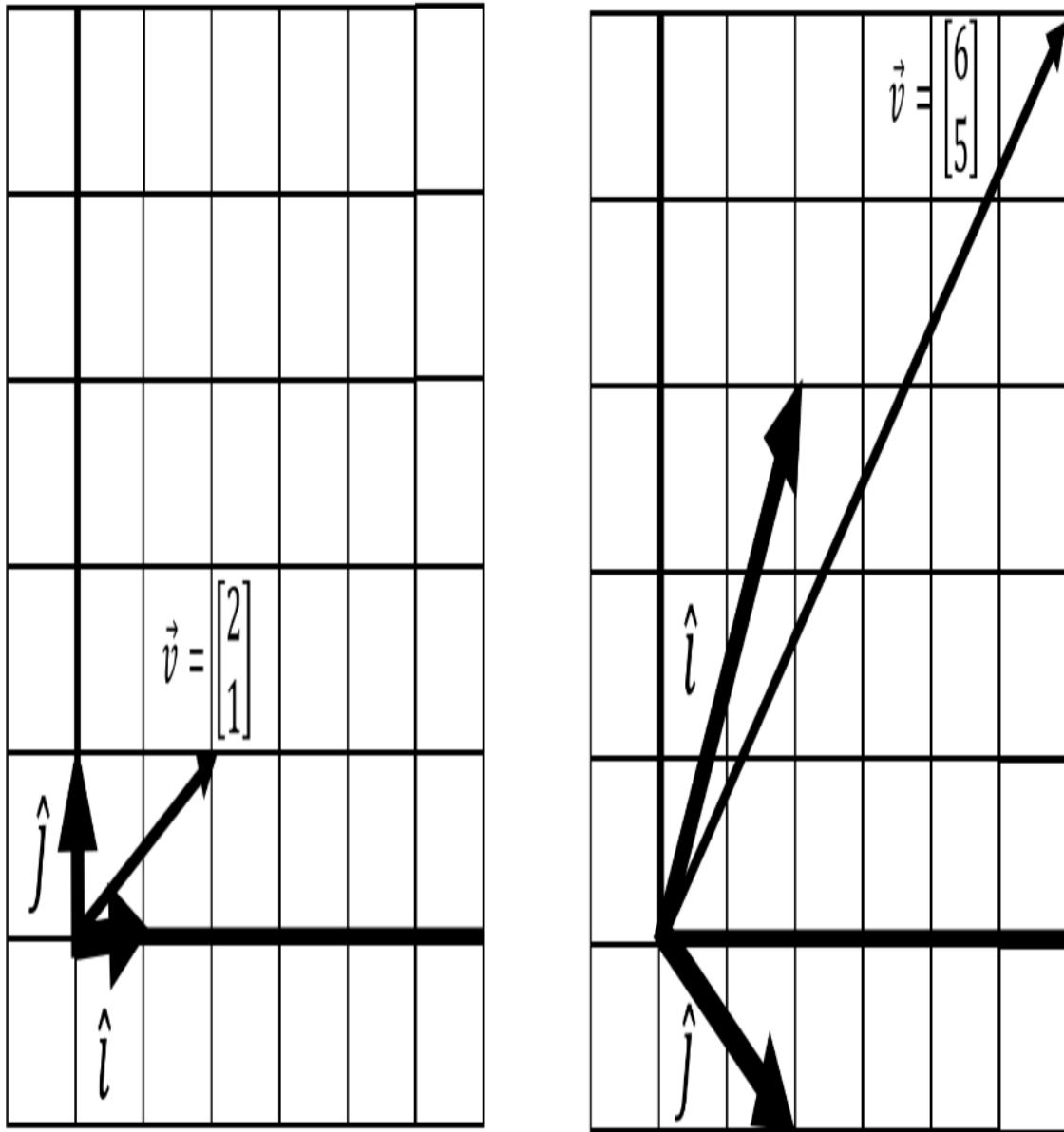


Figure 4-18. A linear transformation that does a rotation, sheer, and flipping of space

Example 4-10.

```
from numpy import array
```

```
# Declare i-hat and j-hat
i_hat = array([2, 3])
j_hat = array([2, -1])
```

```

# compose basis matrix using i-hat and j-hat
# also need to transpose rows into columns
basis = array([i_hat, j_hat]).transpose()

# declare vector v 0
v = array([2,1])

# create new vector
# by transforming v with dot product
new_v = basis.dot(v)

print(new_v) # [6, 5]

```

A lot has happened here. Not only did we scale \hat{i} and \hat{j} and elongate vector \vec{v} . We actually sheered, rotated, and flipped space too. You know space was flipped space when \hat{i} and \hat{j} change places in their clockwise orientation, and we will learn how to detect this with determinants later in this chapter.

BASIS VECTORS IN 3D AND BEYOND

You might be wondering how we think of vector transformations in 3 dimensions or more. The concept of basis vectors extends quite nicely. If I have a 3-dimensional vector space, then I have basis vectors \hat{i} , \hat{j} , and \hat{k} . I just keep adding more letters from the alphabet for each new dimension (Figure 4-19).

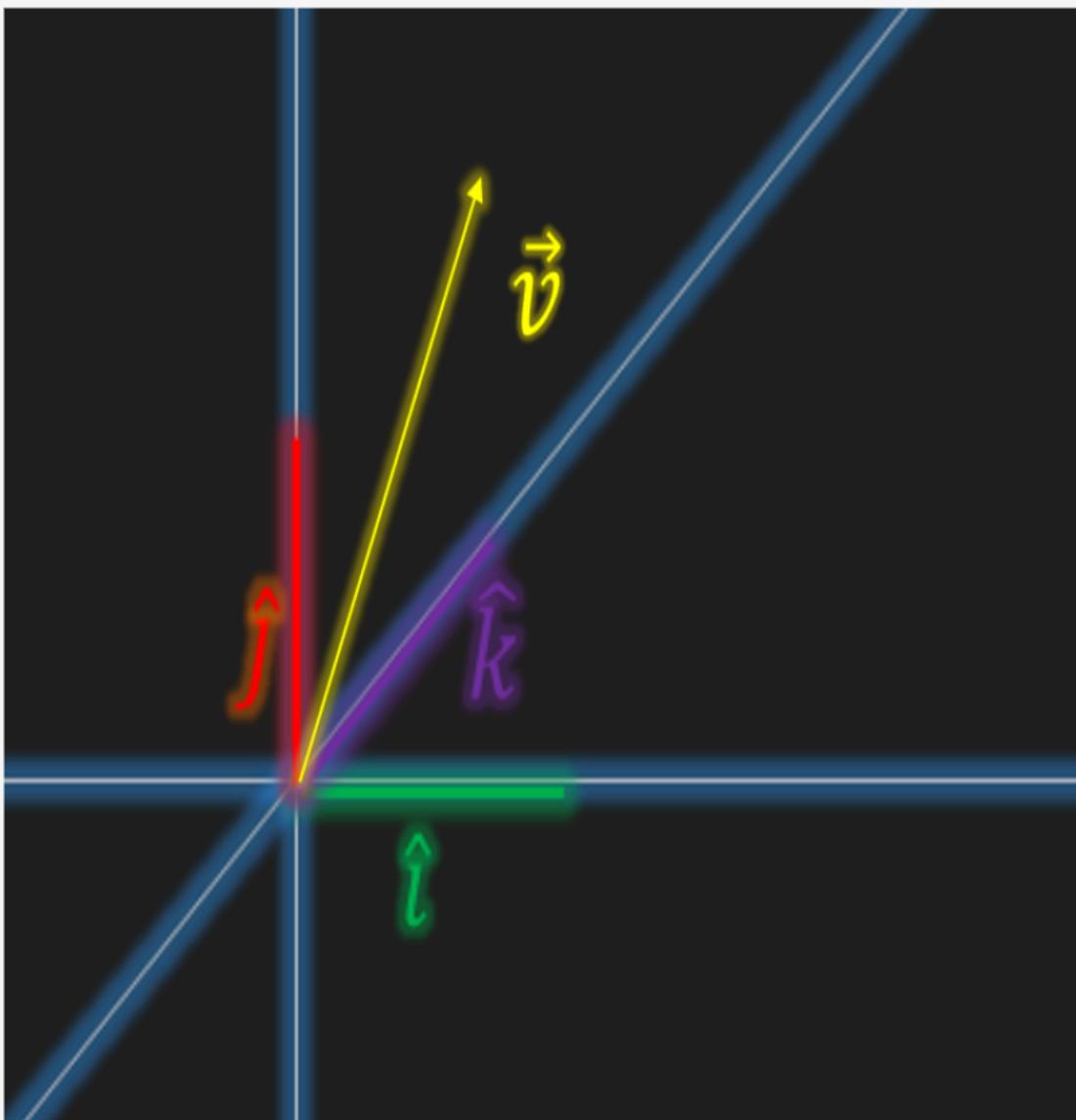


Figure 4-19. 3D basis vector

Something else that is worth pointing out is some linear transformations can shift a vector space into fewer or more dimensions, and that is

exactly what nonsquare matrices will do (where number of rows and columns are not equal). In the interest of time we cannot explore this. But [3Blue1Brown explains and animates this concept beautifully](#).

Matrix Multiplication

We learned how to multiply a vector and a matrix, but what exactly does multiplying two matrices accomplish? Think of **matrix multiplication** as applying multiple transformations to a vector space. Each transformation is like a function, where we apply the inner-most first and then apply each subsequent transformation outwards.

Here is how we apply a rotation and then a shear to any vector \vec{v} with value $[x \ y]$.

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

We can actually consolidate these two transformations by using this formula, applying one transformation onto the last. You multiply and add each row from the first matrix to each respective column of the second matrix, in an “over-and-down! over-and-down!” pattern.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dy & cf + dh \end{bmatrix}$$

So we can actually consolidate these two separate transformations (rotation and shear) into a single transformation.

$$\begin{aligned} & \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ &= \begin{bmatrix} (1)(0) + (1)(1) & (-1)(1) + (1)(0) \\ (0)(0) + (1)(1) & (0)(-1) + (1)(0) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \end{aligned}$$

$$= \begin{bmatrix} 1 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

To execute this in Python using NumPy, you can combine the two matrices simply using the `matmul()` or `@` operator ([Example 4-11](#)). We will then turn around and use this consolidated transformation and apply it to a vector $[1 2]$.

USING DOT() VS MATMULT() AND @

In general, you want to prefer `matmul()` and its shorthand `@` to combine matrices rather than the `dot()` operator in NumPy. The former generally has a preferable policy for higher-dimensional matrices and how the elements are broadcasted.

To learn more, [this StackOverflow question](#) is a good place to start.

Example 4-11.

```
from numpy import array

# Transformation 1
i_hat1 = array([0, 1])
j_hat1 = array([-1, 0])
transform1 = array([i_hat1, j_hat1]).transpose()

# Transformation 2
i_hat2 = array([1, 0])
j_hat2 = array([1, 1])
transform2 = array([i_hat2, j_hat2]).transpose()

# Combine Transformations
combined = transform2 @ transform1

# Test
print("COMBINED MATRIX:\n {}".format(combined))

v = array([1, 2])
print(combined.dot(v)) # [-1 1]
```

Note that we could have also applied each transformation individually to vector \vec{v} and still have gotten the same result. If you replace the last line with

these three lines applying each transformation ([Example 4-12](#)), you will still get $[-1 \ 1]$ on that new vector.

Example 4-12.

```
rotated = transform1.dot(v)
sheered = transform2.dot(rotated)
print(sheered) # [-1 1]
```

Note that the order you apply each transformation matters! If we were to apply `transformation1` on `transformation2`, we are going to get a different result of $[-2 \ 3]$ as calculated in [Example 4-13](#). So matrix dot products are not commutative, meaning you cannot flip the order and expect the same result!

Example 4-13.

```
from numpy import array

# Transformation 1
i_hat1 = array([0, 1])
j_hat1 = array([-1, 0])
transform1 = array([i_hat1, j_hat1]).transpose()

# Transformation 2
i_hat2 = array([1, 0])
j_hat2 = array([1, 1])
transform2 = array([i_hat2, j_hat2]).transpose()

# Combine Transformations, apply sheer first and then rotation
combined = transform1 @ transform2

# Test
print("COMBINED MATRIX:\n {}".format(combined))

v = array([1, 2])
print(combined.dot(v)) # [-2 3]
```

Think of each transformation as a function, and we apply them from the inner-most to outer-most just like nested function calls.

LINEAR TRANSFORMATIONS IN PRACTICE

You might be wondering what do all these linear transformations and matrices have to do with data science and machine learning. The answer is everything! From importing data to numerical operations with linear regression, logistic regression, and neural networks, linear transformations are the heart of mathematically manipulated data.

In practice however, you will rarely take the time to geometrically visualize your data as vector spaces and linear transformations. You will be dealing with too many dimensions to do this productively. But it is good to be mindful of the geometric interpretation just to understand what these contrived-looking numerical operations do! Otherwise you just memorizing numerical operation patterns without any context. It also makes new linear algebra concepts like determinants more obvious.

Determinants

When we perform linear transformations, we sometimes “expand” or “squish” space and the degree this happens can be helpful. Take a sampled area from the vector space in [Figure 4-20](#): what happens to it after we scale \hat{i} and \hat{j} ?

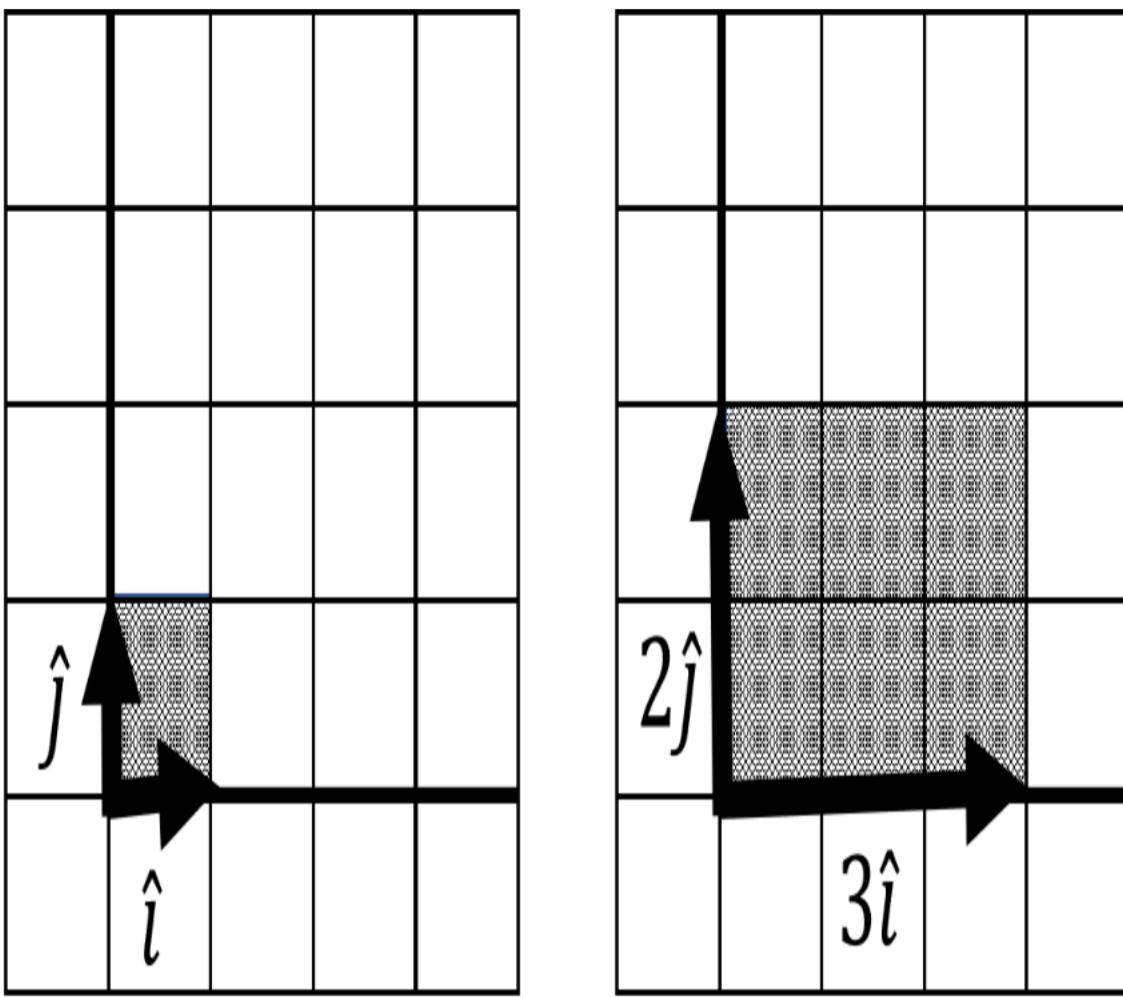


Figure 4-20. A determinant measures how a linear transformation scales an area

Note it increases in area by a factor of 6.0, and this factor is known as a **determinant**. Determinants describe how much a sampled area in a vector space changes in scale with linear transformations, and this can provide helpful information about the transformation.

Example 4-14 shows how to calculate this determinant in Python.

Example 4-14.

```
from numpy.linalg import det
from numpy import array

i_hat = array([3, 0])
j_hat = array([0, 2])

basis = array([i_hat, j_hat]).transpose()
```

```
determinant = det(basis)  
  
print(determinant) # prints 6.0
```

Simple shears and rotations should not affect the determinant as the area will not change. [Figure 4-21](#) and [Example 4-15](#) shows a simple shear and the determinant remains a factor 1.0, showing it is unchanged.

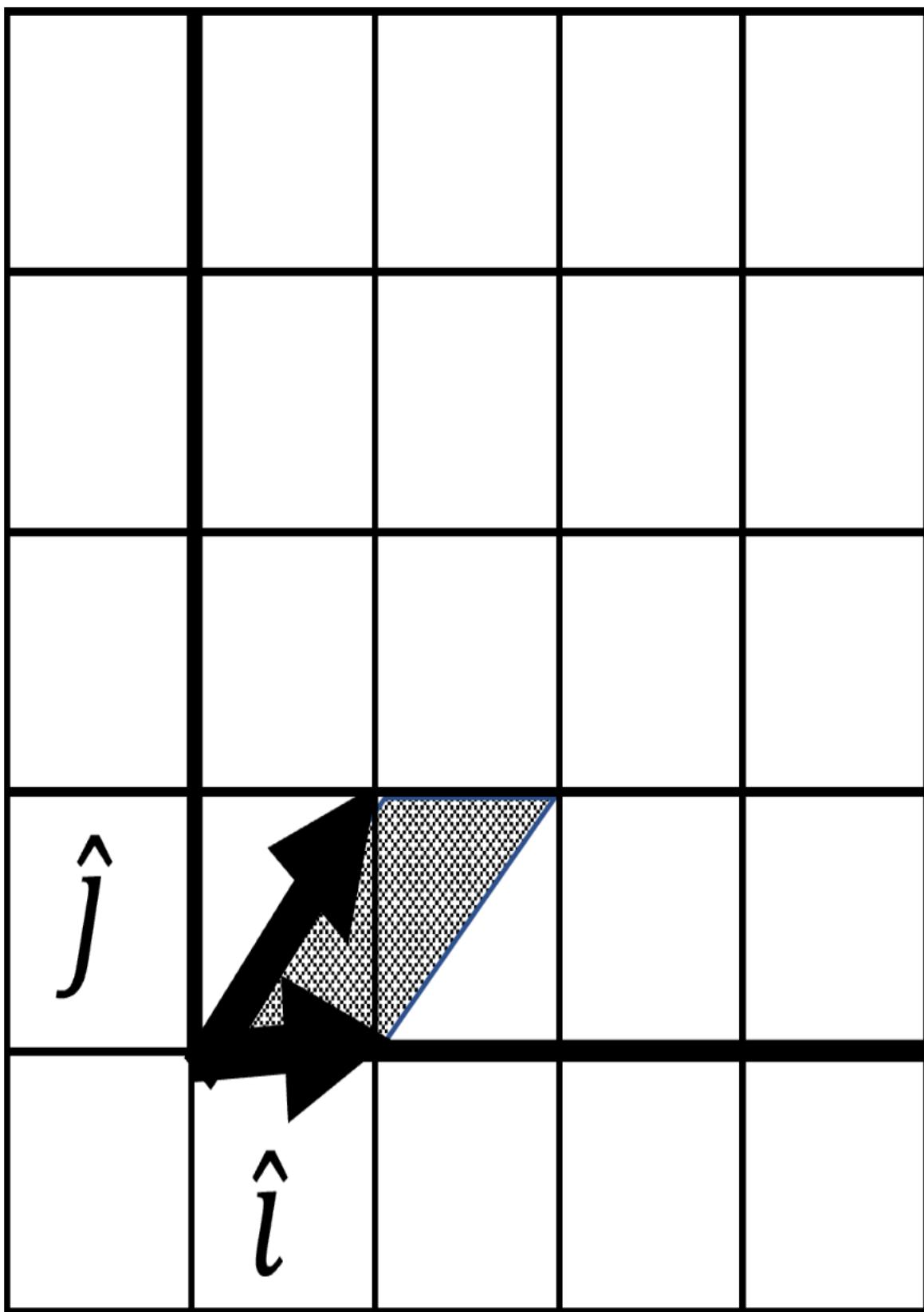


Figure 4-21. A simple shear does not change the determinant

Example 4-15.

```
from numpy.linalg import det
from numpy import array

i_hat = array([1, 0])
j_hat = array([1, 1])

basis = array([i_hat, j_hat]).transpose()

determinant = det(basis)

print(determinant) # prints 1.0
```

But scaling will increase or decrease the determinant, as that will increase/decrease the sampled area. When the orientation flips (\hat{i}, \hat{j} swap clockwise positions) then the determinant will be negative. [Figure 4-22](#) and [Example 4-16](#) is a determinant showing a transformation that not only scaled but also flipped the orientation of the vector space.

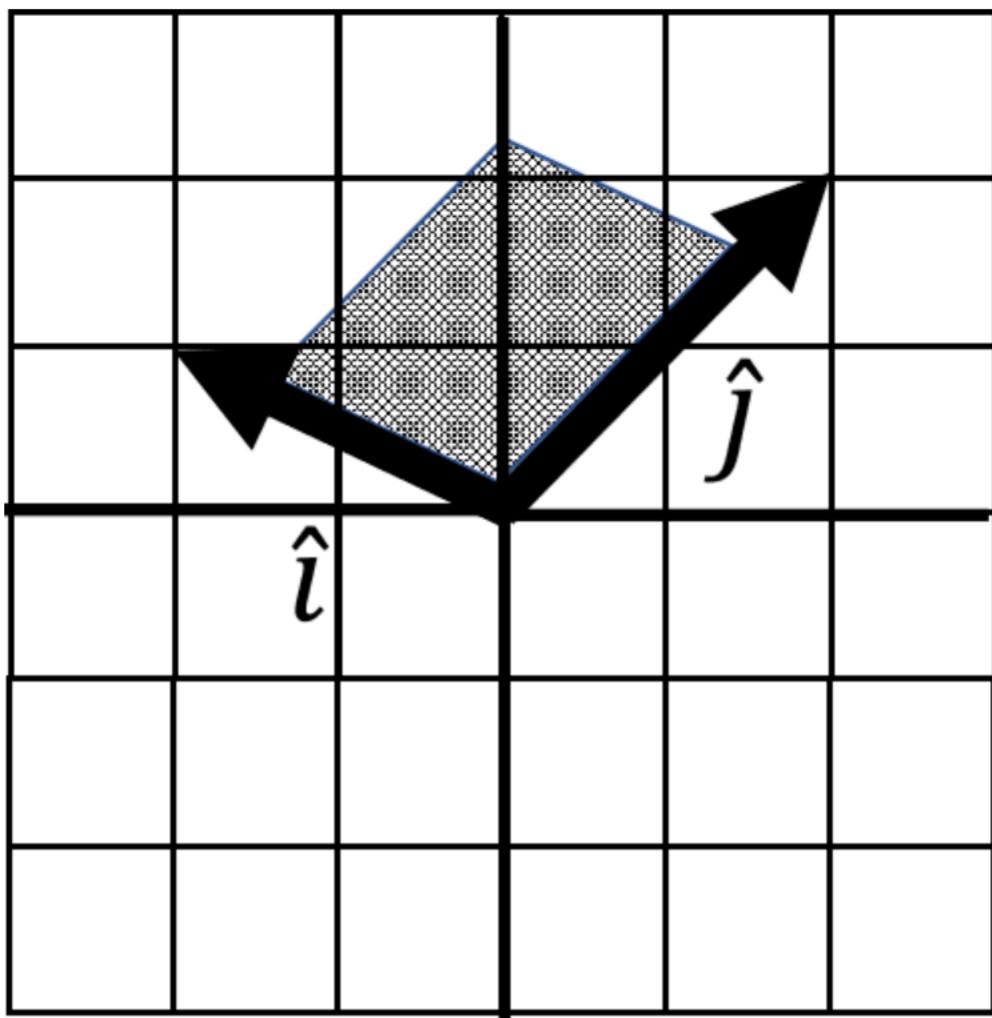


Figure 4-22. A determinant on a flipped space is negative

Example 4-16.

```

from numpy.linalg import det
from numpy import array

i_hat = array([-2, 1])
j_hat = array([1, 2])

basis = array([i_hat, j_hat]).transpose()

determinant = det(basis)

print(determinant) # prints -5.0

```

Because this determinant is negative, we quickly see that the orientation has flipped.

THE RIGHT HAND RULE

The determinant can extend to 3 or more dimensions. It becomes a matter of visualizing a sampled *volume* scaling, rotating, shearing, and flipping as visualized in [Figure 4-23](#).

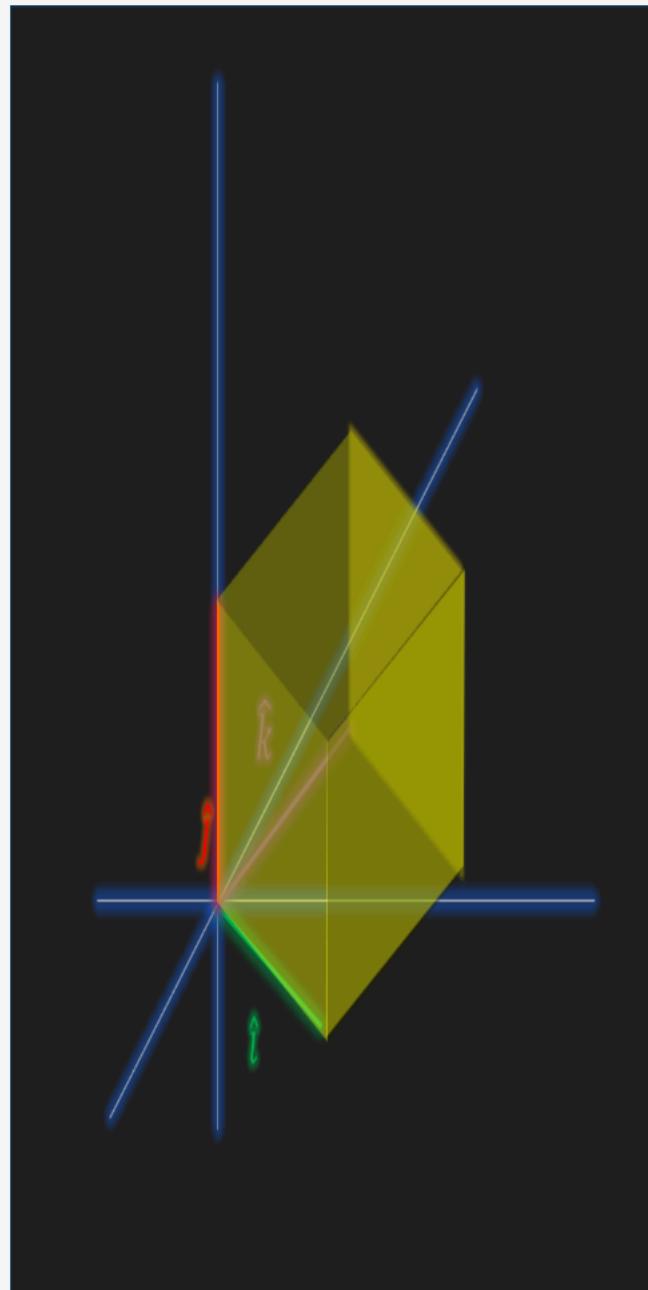


Figure 4-23. The right hand rule

To figure out if a 3D space has flipped, use the **right hand rule**. If you cannot relatively orient \hat{i} , \hat{j} , and \hat{k} like my hand above the orientation has flipped, and you should have a negative determinant.

But by far, the most critical piece of information the determinant tells you is whether the transformation is linearly dependent. If you have a determinant of 0 that means all of space has been squished into a lesser dimension.

In [Figure 4-24](#) we see two linearly dependent transformations, where a 2D space is compressed into 1 dimension and a 3D space is compressed into 2 dimensions. The area and volume respectively in both cases are 0!

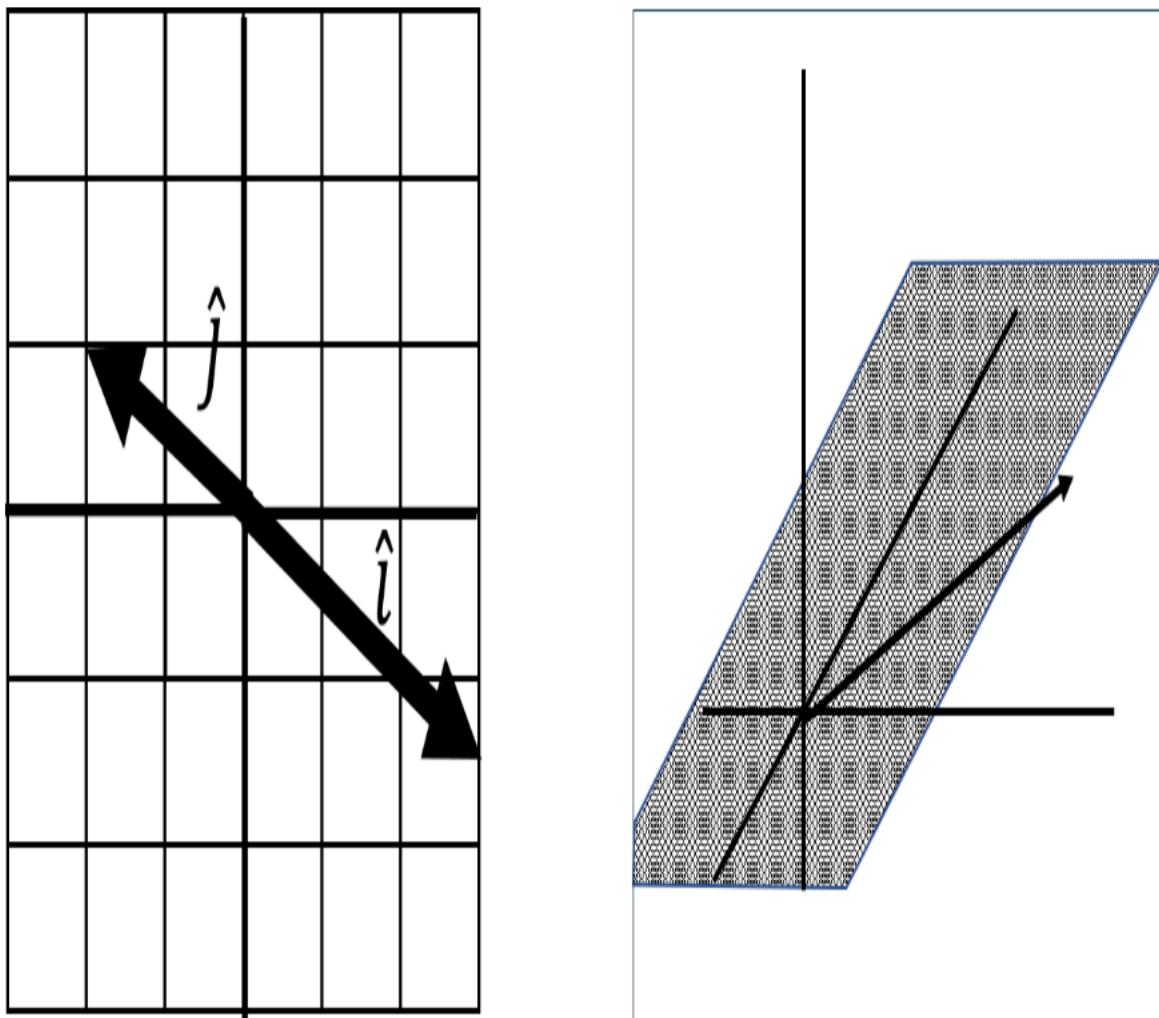


Figure 4-24. Linear dependence in 2D and 3D

Example 4-17 shows the code for the 2D example above squishing an entire 2D space into a single 1-dimensional number line.

Example 4-17.

```
from numpy.linalg import det
from numpy import array

i_hat = array([-2, 1])
j_hat = array([3, -1.5])

basis = array([i_hat, j_hat]).transpose()

determinant = det(basis)

print(determinant) # prints 0.0
```

So testing for a 0 determinant is highly helpful to determine if a transformation has linear dependence. When you encounter this you will likely find a difficult or unsolvable problem on your hands.

Systems of Equations and Inverse Matrices

One of the basic use cases for linear algebra is solving systems of equations. It is also a good application to learn about inverse matrices. Let's say you are provided with the following equations and you need to solve for x , y , and z .

$$4x + 2y + 4z = 44$$

$$5x + 3y + 7z = 56$$

$$9x + 3y + 6z = 72$$

You can try manually experimenting with different algebraic operations to isolate the three variables, but if you want a computer to solve it you will need to express this problem in terms of matrices as shown below. Extract the coefficients into matrix A , the values on the right-side of the equation into matrix B , and the unknown variables into matrix X .

$$A = \begin{vmatrix} 4 & 2 & 4 \\ 5 & 3 & 7 \\ 9 & 3 & 6 \end{vmatrix}$$

$$B = \begin{vmatrix} 44 \\ 56 \\ 72 \end{vmatrix}$$

$$X = \begin{vmatrix} x \\ y \\ z \end{vmatrix}$$

The function for a linear system of equations is $AX = B$, we need to transform matrix A with some other matrix X that will result in matrix B .

$$AX = B$$

$$\begin{vmatrix} 4 & 2 & 4 \end{vmatrix} \cdot \begin{vmatrix} x \\ y \\ z \end{vmatrix} = \begin{vmatrix} 44 \\ 56 \\ 72 \end{vmatrix}$$

We need to “undo” A so we can isolate X and get the values for x , y , and z . The way you undo A is to take the inverse of A denoted by A^{-1} and apply it to A via matrix multiplication. We can express this algebraically below:

$$AX = B$$

$$A^{-1}AX = A^{-1}B$$

$$X = A^{-1}B$$

To calculate the inverse of matrix A , we should probably use a computer rather than searching for solutions by hand using Gaussian elimination, which we will not venture into in this book ([PatrickJMT has some great videos on YouTube](#)). Here is the inverse of matrix A .

$$A^{-1} = \begin{vmatrix} -\frac{1}{2} & 0 & \frac{1}{3} \\ 5.5 & -2 & \frac{4}{3} \\ -2 & 1 & \frac{1}{3} \end{vmatrix}$$

Note when we matrix multiply A^{-1} against A it will create an **identity matrix**, creating a matrix of all zeroes except for 1's in the diagonal. The identity matrix is the linear algebra equivalent of multiplying by “1”, meaning it essentially has no effect and will effectively isolate values for x , y , and z .

$$A^{-1} = \begin{vmatrix} -\frac{1}{2} & 0 & \frac{1}{3} \\ 5.5 & -2 & \frac{4}{3} \\ -2 & 1 & \frac{1}{3} \end{vmatrix}$$

$$A = \begin{vmatrix} 4 & 2 & 4 \\ 5 & 3 & 7 \\ 9 & 3 & 6 \end{vmatrix}$$

$$A^{-1}A = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

To see this identity matrix in action in Python, you will probably want to use SymPy here instead of NumPy. The floating point decimals in NumPy will not make the identity matrix so obvious, but doing it symbolically in [Example 4-18](#) we will see a clean, symbolic output. Note that to do matrix multiplication in SymPy we use the asterisk * rather than @.

Example 4-18. Using SymPy to study the inverse and identity matrix

```
from sympy import *
```

```
# 4x + 2y + 4z = 44
# 5x + 3y + 7z = 56
# 9x + 3y + 6z = 72
```

```
A = Matrix([
    [4, 2, 4],
```

```

[5, 3, 7],
[9, 3, 6]
])

# dot product between A and its inverse
# will produce identity function
inverse = A.inv()
identity = inverse * A

# prints Matrix([[-1/2, 0, 1/3], [11/2, -2, -4/3], [-2, 1, 1/3]])
print("INVERSE: {}".format(inverse))

# prints Matrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
print("IDENTITY: {}".format(identity))

```

In practice though, the lack of floating point precision will not affect our answers too badly, so using NumPy should be fine to solve for X . Example 4-19 shows a solution with NumPy.

Example 4-19. Using NumPy to solve a system of equations

```

from numpy import array
from numpy.linalg import inv

# 4x + 2y + 4z = 44
# 5x + 3y + 7z = 56
# 9x + 3y + 6z = 72

A = array([
    [4, 2, 4],
    [5, 3, 7],
    [9, 3, 6]
])

B = array([
    44,
    56,
    72
])

X = inv(A).dot(B)

print(X) # [ 2. 34. -8.]

```

So $x = 2$, $y = 34$, and $z = -8$. Since I did use SymPy in this problem, Example 4-20 shows the full solution in SymPy as an alternative to NumPy.

Example 4-20. Using SymPy to solve a system of equations

```
from sympy import *

# 4x + 2y + 4z = 44
# 5x + 3y + 7z = 56
# 9x + 3y + 6z = 72

A = Matrix([
    [4, 2, 4],
    [5, 3, 7],
    [9, 3, 6]
])

B = Matrix([
    44,
    56,
    72
])

X = A.inv() * B

print(X) # Matrix([[2], [34], [-8]])
```

Here is the solution in mathematical notation:

$$A^{-1}B = X$$
$$\begin{vmatrix} -\frac{1}{2} & 0 & \frac{1}{3} \end{vmatrix} \begin{vmatrix} 44 \end{vmatrix} = \begin{vmatrix} x \end{vmatrix}$$
$$\begin{vmatrix} 5.5 & -2 & \frac{4}{3} \end{vmatrix} \begin{vmatrix} 56 \end{vmatrix} = \begin{vmatrix} y \end{vmatrix}$$
$$\begin{vmatrix} -2 & 1 & \frac{1}{3} \end{vmatrix} \begin{vmatrix} 72 \end{vmatrix} = \begin{vmatrix} z \end{vmatrix}$$
$$\begin{vmatrix} 2 \end{vmatrix} = \begin{vmatrix} x \end{vmatrix}$$
$$\begin{vmatrix} 34 \end{vmatrix} = \begin{vmatrix} y \end{vmatrix}$$
$$\begin{vmatrix} -8 \end{vmatrix} = \begin{vmatrix} z \end{vmatrix}$$

Hopefully this gave you an intuition for inverse matrices and how they can be used to solve a system of equations.

SYSTEMS OF EQUATIONS IN LINEAR PROGRAMMING

This method of solving systems of equations is used for linear programming as well, where inequalities define constraints and an objective is minimized/maximized.

[PatrickJMT has a lot of good videos on Linear Programming](#)

In practicality you should rarely find it necessary to calculate inverse matrices by hand and can have a computer do it for you. But if you have a need or are curious, you will want to learn about Gaussian elimination. [PatrickJMT on YouTube](#) has a number of videos demonstrating Gaussian elimination.

Eigenvectors and Eigenvalues

Matrix decomposition is breaking up a matrix into its basic components, much like factoring numbers (e.g. 10 can be factored to 2×5).

Matrix decomposition is helpful for tasks like finding inverse matrices, calculating determinants, as well as linear regression. There are many ways to decompose a matrix depending on your task. In [Chapter 5](#) we will use a matrix decomposition technique, QR decomposition, to perform a linear regression. But in this chapter let's focus on a common method called eigendecomposition, which is often used for machine learning and Principal Component Analysis (PCA). At this level we do not have the bandwidth to dive into each of these applications. For now, just know eigendecomposition is helpful for breaking up a matrix into components that are easier to work with in different machine learning tasks.

In eigendecomposition, there are two components: the eigenvalues denoted by lambda λ and eigenvector by v shown in [Figure 4-25](#).

$$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix}$$



$$\begin{bmatrix} -0.464 \\ 6.464 \end{bmatrix}$$

$$\begin{bmatrix} 0.806 & 0.343 \\ 0.59 & -0.939 \end{bmatrix}$$

λ

v

Figure 4-25. The eigenvector and eigenvalues

If we have a square matrix A , it has the following eigenvalue equation:

$$Av = \lambda v$$

If A is the original matrix, it is composed of eigenvector v and eigenvalue λ . There is one eigenvector and eigenvalue for each dimension of the parent matrix, and not all matrices can be decomposed into an eigenvector and eigenvalue. Sometimes complex (imaginary) numbers will even result.

Example 4-21 is how we calculate eigenvectors and eigenvalues in NumPy for a given matrix A :

Example 4-21. Performing eigendecomposition in NumPy

```
from numpy import array, diag
from numpy.linalg import eig, inv

A = array([
    [1, 2],
    [4, 5]
])

eigenvals, eigenvecs = eig(A)

print("EIGENVALUES")
print(eigenvals)
print("\nEIGENVECTORS")
print(eigenvecs)

"""
EIGENVALUES
[-0.46410162  6.46410162]

EIGENVECTORS
[[-0.80689822 -0.34372377]
 [ 0.59069049 -0.9390708 ]]
"""


```

So how do we rebuild matrix A from the eigenvectors and eigenvalues?
Recall this formula:

$$Av = \lambda v$$

We need to make a few tweaks to the formula to reconstruct A :

$$A = Q\Lambda Q^{-1}$$

In this new formula, Q is the eigenvectors, Λ is the eigenvalues in diagonal form, and Q^{-1} is the inverse matrix of Q . Diagonal form means the vector is padded into a matrix of zeroes, and occupies the diagonal line in a similar pattern to an identity matrix.

Example 4-22 brings the example full circle in Python, starting with decomposing the matrix and then recomposing it.

Example 4-22. Decomposing and recomposing a matrix in NumPy

```
from numpy import array, diag
from numpy.linalg import eig, inv
```

```
A = array([
    [1, 2],
    [4, 5]
])

eigenvals, eigenvecs = eig(A)

print("EIGENVALUES")
print(eigenvals)
print("\nEIGENVECTORS")
print(eigenvecs)

print("\nREBUILD MATRIX")
Q = eigenvecs
R = inv(Q)

L = diag(eigenvals)
B = Q @ L @ R

print(B)
```

```
"""
EIGENVALUES
[-0.46410162  6.46410162]
```

```
EIGENVECTORS
[[-0.80689822 -0.34372377]
 [ 0.59069049 -0.9390708]]
```

```
REBUILD MATRIX
[[1. 2.]
 [4. 5.]]
"""
```

As you can see above, the matrix we rebuilt is the one we started with.

Conclusion

Linear algebra can be maddeningly abstract and it is full of mysteries and ideas to ponder. You may find the whole topic is one big rabbit hole, and you

would be right! However it is a good idea to continue being curious about it if you want to have a long, successful data science career. It is the foundation for statistical computing, machine learning, and other applied data science areas. Ultimately, it is the foundation for computer science in general. You can certainly get away with not knowing it for awhile but you will encounter limitations in your understanding at some point.

You may wonder how these ideas are practical as they may feel theoretical. Do not worry, we will see some practical applications throughout this book. But the theory and geometric interpretations are important to have intuition when you work with data, and by understanding linear transformations visually you are prepared to take on more advanced concepts that may be thrown at you later in your pursuits.

If you want to learn more about linear programming, there is no better place than [3Blue1Brown's YouTube playlist "Essence of Linear Algebra"](#). The [linear algebra videos from PatrickJMT](#) are helpful as well.

If you want to get more comfortable with NumPy, the O'Reilly book [Python for Data \(2nd edition\)](#) by Wes McKinney is a recommended read. It does not focus much on linear algebra but it does provide practical instruction on using NumPy, Pandas, and Python on datasets.

Exercises

1. Vector \vec{v} has a value of $[1 \ 2]$ but then a transformation happens. \hat{i} lands at $[2 \ 0]$ and \hat{j} lands at $[0 \ 1.5]$. Where does \vec{v} land?
2. Vector \vec{v} has a value of $[1 \ 2]$ but then a transformation happens. \hat{i} lands at $[-2 \ 1]$ and \hat{j} lands at $[1 \ -2]$. Where does \vec{v} land?
3. A transformation lands \hat{i} lands at $[1 \ 0]$ and \hat{j} lands at $[2 \ 2]$. What is the determinant of this transformation?
4. Can two or more linear transformations be done in single linear transformation? Why or why not?

5. Solve the system of equations for x , y , and z .

$$3x + 1y + 0z = 54$$

$$2x + 4y + 1z = 12$$

$$3x + 1y + 8z = 6$$

1. Is the matrix below linearly dependent? Why or why not?

$$\begin{bmatrix} 2 & 1 \\ 6 & 3 \end{bmatrix}$$

Chapter 5. Linear Regression

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at thomasnield@live.com.

One of the most practical techniques in data analysis is fitting a line through observed data points to show a relationship between two or more variables. A **regression** attempts to fit a function to observed data to make predictions on new data. A **linear regression** fits a straight line to observed data, attempting to demonstrate a linear relationship between variables and make predictions on new data yet to be observed.

It might make more sense to see a picture rather than read a description of linear regression. Here is an example of a linear regression in [Figure 5-1](#).

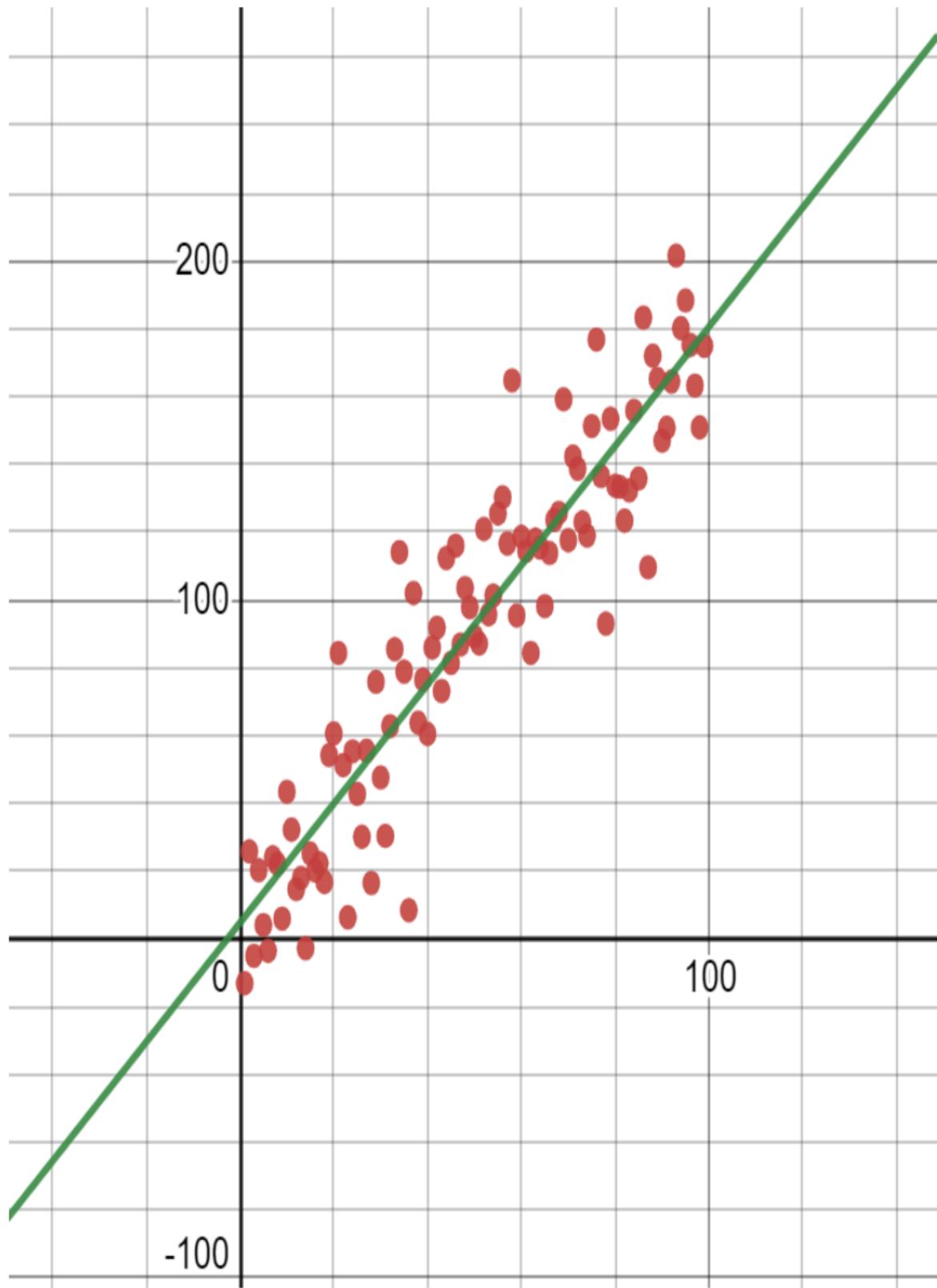


Figure 5-1. Example of a linear regression, which fits a line to observed data

Linear regression is a workhorse of data science and statistics, and not only applies concepts we learned in previous chapters but sets up new foundations for later topics like neural networks and logistic regression. This relatively simple technique has been around for over 200 years, and contemporarily is branded as a form of machine learning.

Machine learning practitioners are less likely to analyze their linear regression models in depth, and simply do a quick train-test split of the data. Statisticians, however, will question the linear regression more deeply, analyzing the prediction intervals and correlation for statistical significance. We will cover both schools of thought so readers can bridge the ever-widening gap between the two, and thus find themselves best equipped to wear both hats.

WAIT, MACHINE LEARNING IS JUST REGRESSION?

“Machine learning” is generally another name for regression models, particularly in the context of supervised learning. This is why linear regression is considered a form of machine learning. Confusingly, statisticians may refer to their regression models as *statistical learning*, whereas data science and machine learning professionals will call their models *machine learning*.

While supervised learning is often regression, unsupervised machine learning is more about clustering and anomaly detection. Reinforcement learning often pairs supervised machine learning with simulation to rapidly generate data. Of course, the data is as only good as the simulation and the simulation-to-reality gap is always an issue.

A Basic Linear Regression

I want to study the relationship between the age of a dog and the number of veterinary visits it had. In a sample we have 10 random dogs, let’s plot them as shown in [Figure 5-2](#).

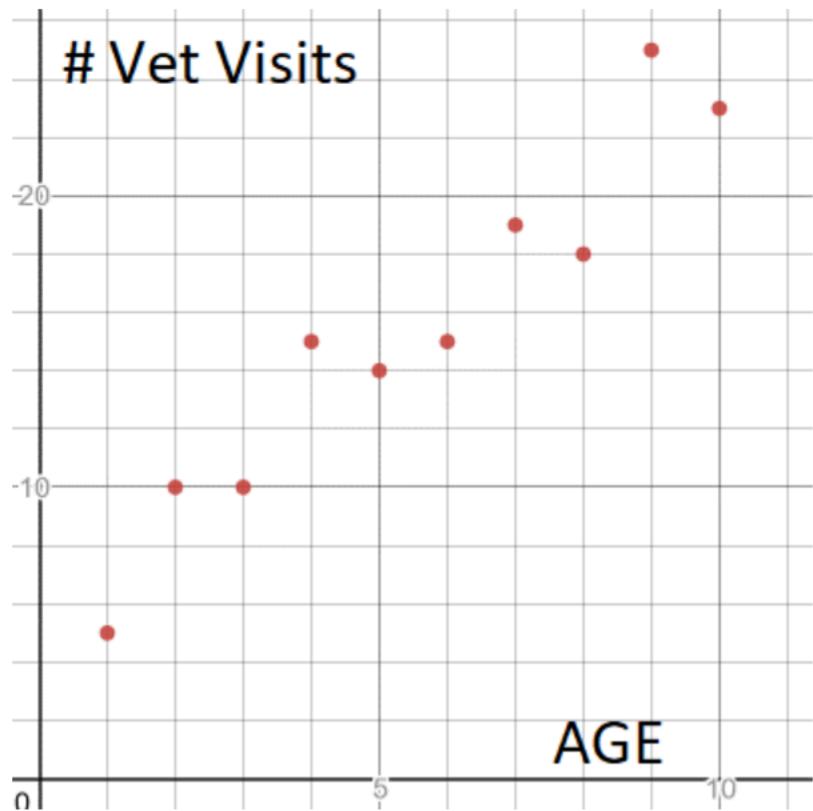


Figure 5-2. Plotting a sample of 10 dogs with their age and number of vet visits

We can clearly see there is a **linear correlation** here, meaning when one of these variables increases/decreases, the other increases/decreases in a roughly proportional amount. We could draw a line through these points to show a correlation like this in [Figure 5-3](#).

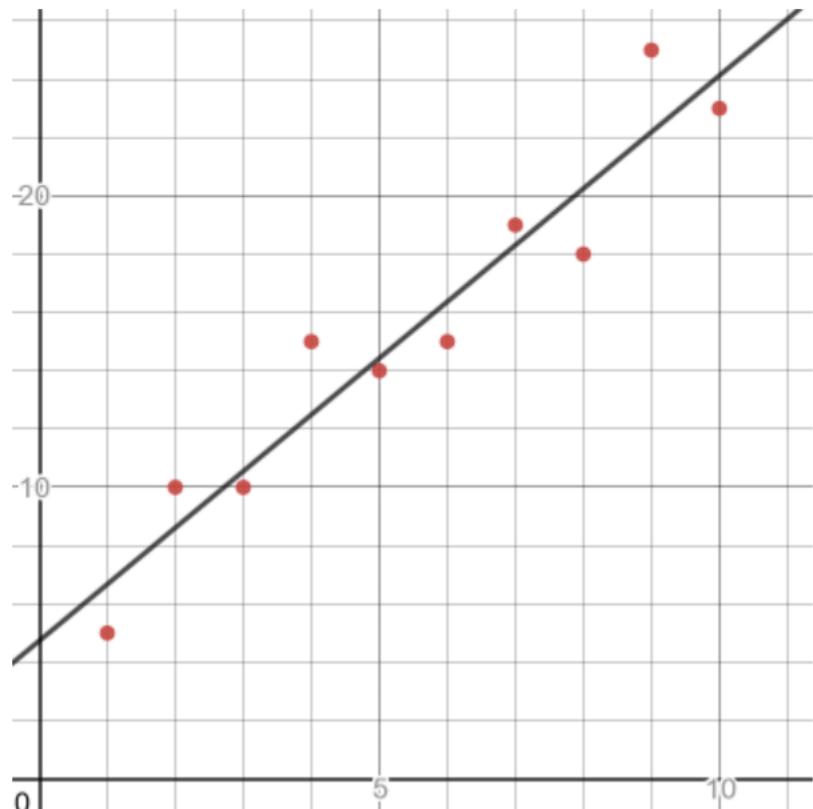


Figure 5-3. Fitting a Line Through Our Data

I will show how to calculate this fitted line later in this chapter. We will also explore how to calculate the quality of this fitted line. For now, let's focus on the benefits of performing a linear regression. It allows us to make predictions on data we have not seen before. I do not have a dog in my sample that is 8.5 years old, but I can look at this line and estimate the dog will have 21 veterinary visits in its life. I just look at the line where $x=8.5$ and I see that $y=21.218$ as shown in Figure 5-4.

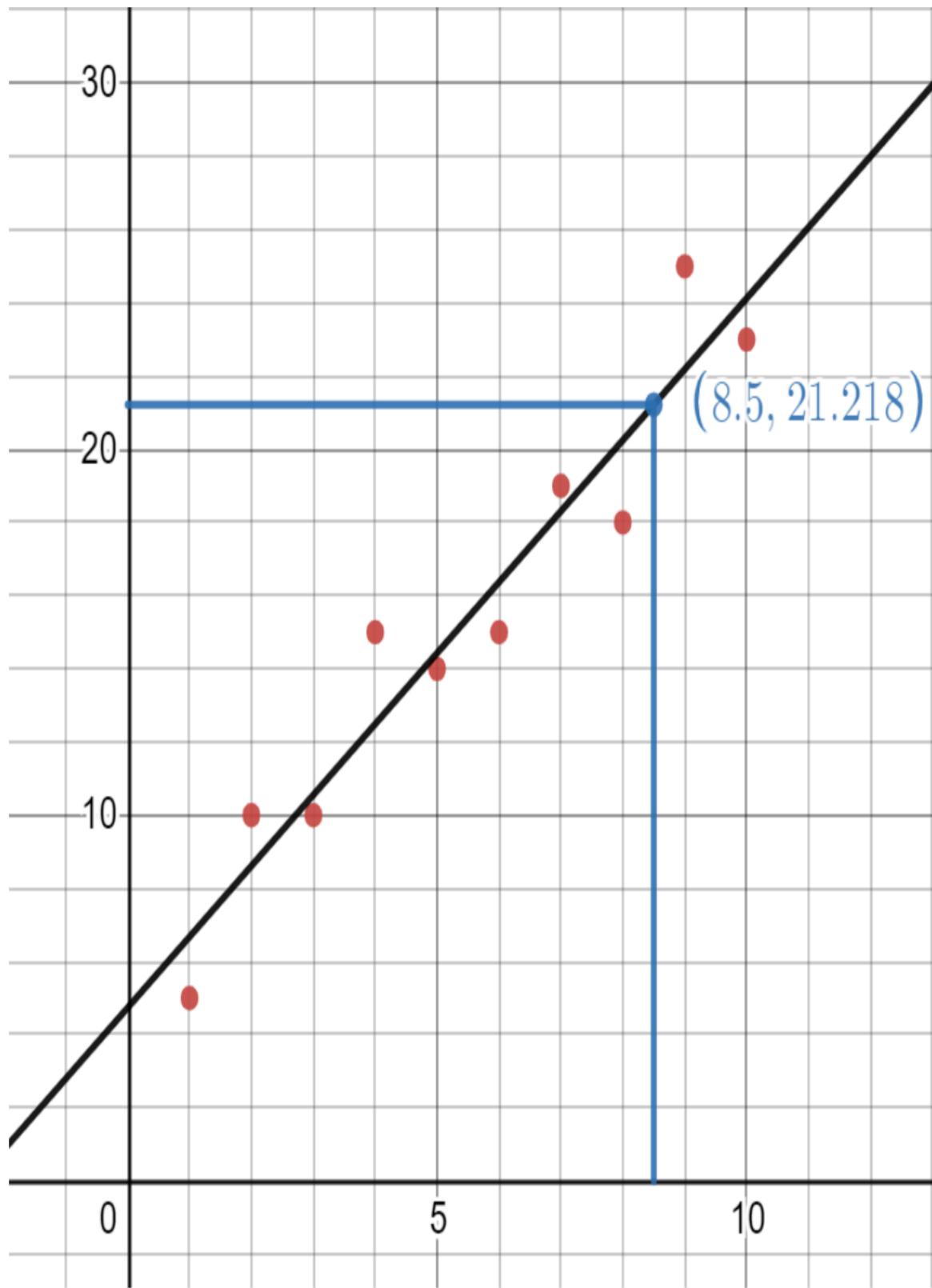


Figure 5-4. Making a prediction using a linear regression, seeing that an 8.5 year old dog is predicted to have about 21.2 vet visits

Now what are the downsides of a linear regression? I cannot expect that every outcome is going to fall EXACTLY on that line. After all, real world data is noisy and never perfect and will not follow a straight line. It may not remotely follow a straight line at all! There is going to be error around that line, where the point will fall above or below the line. We will cover this mathematically when we talk about p-values, statistical significance, and prediction intervals which describes how reliable our linear regression is. Another catch is we should not use the linear regression to make predictions outside the range of data we have, meaning we should not make predictions where $x < 0$ and $x > 10$ because we do not have data outside those values.

DON'T FORGET SAMPLING BIAS!

We should question this data and how it was sampled in order to detect bias. Was this at a single veterinary clinic? Multiple random clinics? Is there self-selection bias by using veterinary data, only polling dogs that visit the vet? If the dogs were sampled in the same geography, can that sway the data? Perhaps dogs in hot desert climates go to vets more for heat exhaustion and snake bites, and this would inflate our veterinary visits in our sample.

As discussed in [Chapter 3](#), it has become fashionable to make data an oracle for truth. However data is simply a sample from a population and we need to practice discernment on how well represented our sample is. Be just as interested (if not more) on where the data comes from and not just what the data says.

Basic Linear Regression with SciPy

We have a lot to learn regarding linear regressions in this chapter, but let's start out with some code to execute what we know so far.

There are plenty of platforms to perform a linear regression, from Excel to Python and R. But we will stick with Python in this book, starting with SciPy to do the work for us. I will show how to build a linear regression “from scratch” later in this chapter so we grasp important concepts like gradient descent and least squares.

Code [Example 5-1](#) is how we use Scikit-Learn to perform a basic, unvalidated linear regression on the sample of 10 dogs. We pull in [this data](#)

using Pandas, convert it into NumPy arrays, perform linear regression using Scikit-Learn, and use plotly to display it in a chart.

Example 5-1. Using Scikit-Learn to do a linear regression

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# Import points
df = pd.read_csv('https://bit.ly/3go0Ant', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)
Y = df.values[:, -1]

# Fit a line to the points
fit = LinearRegression().fit(X, Y)

# m = 1.7867224, b = -16.51923513
m = fit.coef_.flatten()
b = fit.intercept_.flatten()
print("m = {0}".format(m))
print("b = {0}".format(b))

# show in chart
plt.plot(X, Y, 'o') # scatterplot
plt.plot(X, m*X+b) # line
plt.show()
```

First we import the data from [this CSV on GitHub](#). We separate the two columns into X and Y datasets using Pandas. We then `fit()` the `LinearRegression` model to the input X data and the output Y data. We can then get the m and b coefficients that describe our fitted linear function.

In the plot, sure enough you will get a fitted line running through these points shown in [Figure 5-5](#).

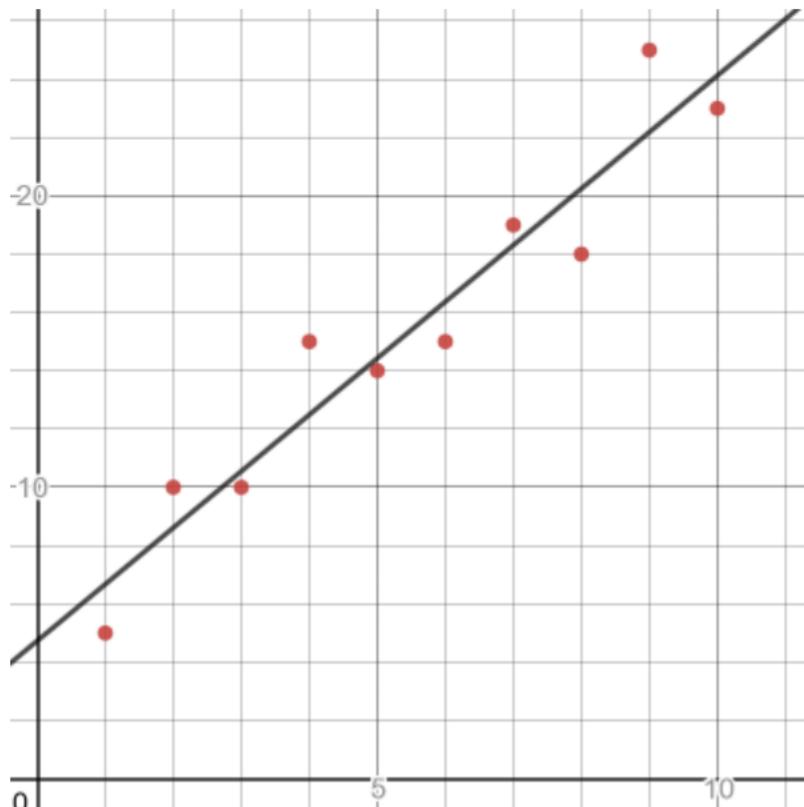


Figure 5-5. SciPy will fit a regression line to your data.

What decides the best fit line to these points? Let's discuss that next.

Residuals and Squared Errors

How do statistics tools like Scikit-Learn come up with a line that fits to these points? Well it comes down to two questions that are fundamental to machine learning training:

- 1) What defines a “best fit”?
- 2) How do we get to that “best fit”?

The first question has a pretty established answer: we minimize the squares, or more specifically the sum of the squared residuals. Let's break that down. Draw any line through the points. The **residual** is the numeric difference between the line and the points, as shown below in Figure 5-6.

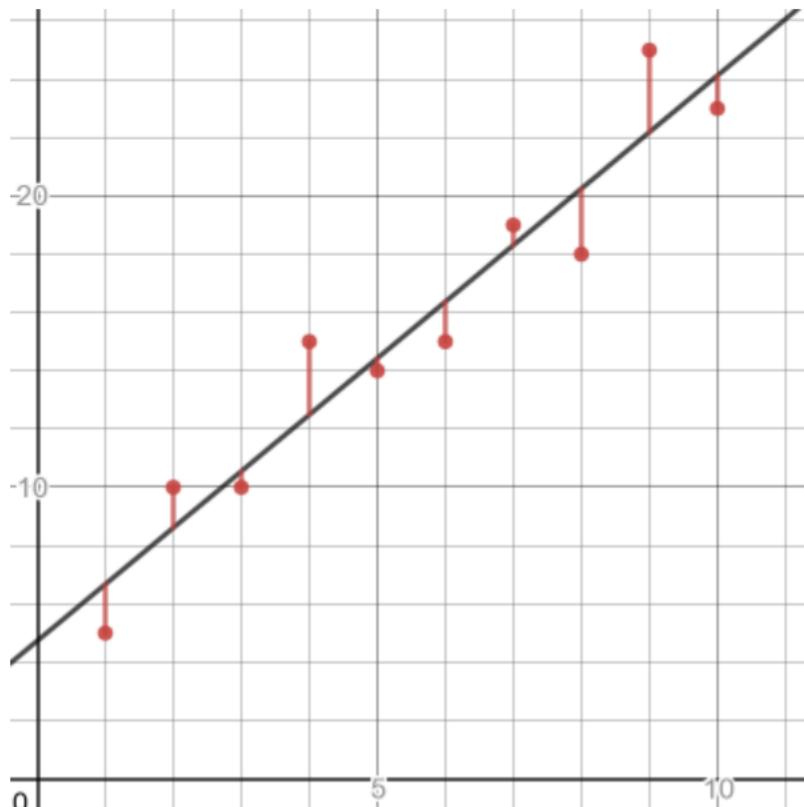


Figure 5-6. The residuals are the differences between the line and the points

Points above the line will have a positive residual, and points below the line will have a negative residual. In other words, it is the subtracted difference between the predicted y-values (derived from the line) and the actual y-values (which came from the data). Another name for residuals are **errors**, because they reflect how wrong our line is in predicting the data.

Let's calculate these differences between these 10 points and the line $y = 1.93939x + 7.73333$ in code [Example 5-2](#).

Example 5-2. Calculating the residuals for a given line and data

```
import pandas as pd
```

```
# Import points
points = pd.read_csv('https://bit.ly/3go0Ant', delimiter=',').itertuples()

# Test with a given line
m = 1.93939
b = 4.73333

# calculate sum of squares
```

```
for p in points:  
    y_actual = p.y  
    y_predict = m*p.x + b  
    residual = y_predict - y_actual  
    print(residual)
```

OUTPUT:

Example 5-3.

```
residual=1.67272  
residual=-1.3878900000000005  
residual=0.5515000000000008  
residual=-2.509109999999997  
residual=0.430279999999998  
residual=1.369669999999993  
residual=-0.6909400000000012  
residual=2.248449999999983  
residual=-2.812160000000002  
residual=1.127229999999973
```

If we are fitting a straight line through our 10 data points, we likely want to minimize these residuals in total so there is as little of a gap as possible between the line and points. But how do we measure the “total”? The best approach is to take the **sum of squares**, which simply squares each residual, or multiplies each residual by itself, and sums them. We take each actual y value and subtract from it the predicted y value taken from the line, then square and sum all those differences.

WHY NOT ABSOLUTE VALUES?

You might wonder why we have to square the residuals before summing them. Why not just add them up without squaring? That will not work because the negatives will cancel out the positives. What if we add the absolute values, where we turn all negative values into positive values? That sounds promising but absolute values are mathematically inconvenient. More specifically, absolute values do not work well with Calculus derivatives which we are going to use later for gradient descent. This is why we choose the squared residuals as our way of totaling the loss.

A visual way to think of it is shown in [Figure 5-7](#), where we overlay a square on each residual and each side is the length of the residual. We sum the area of all these squares, and later we will learn how to find the minimum sum we can achieve by identifying the best m and b .

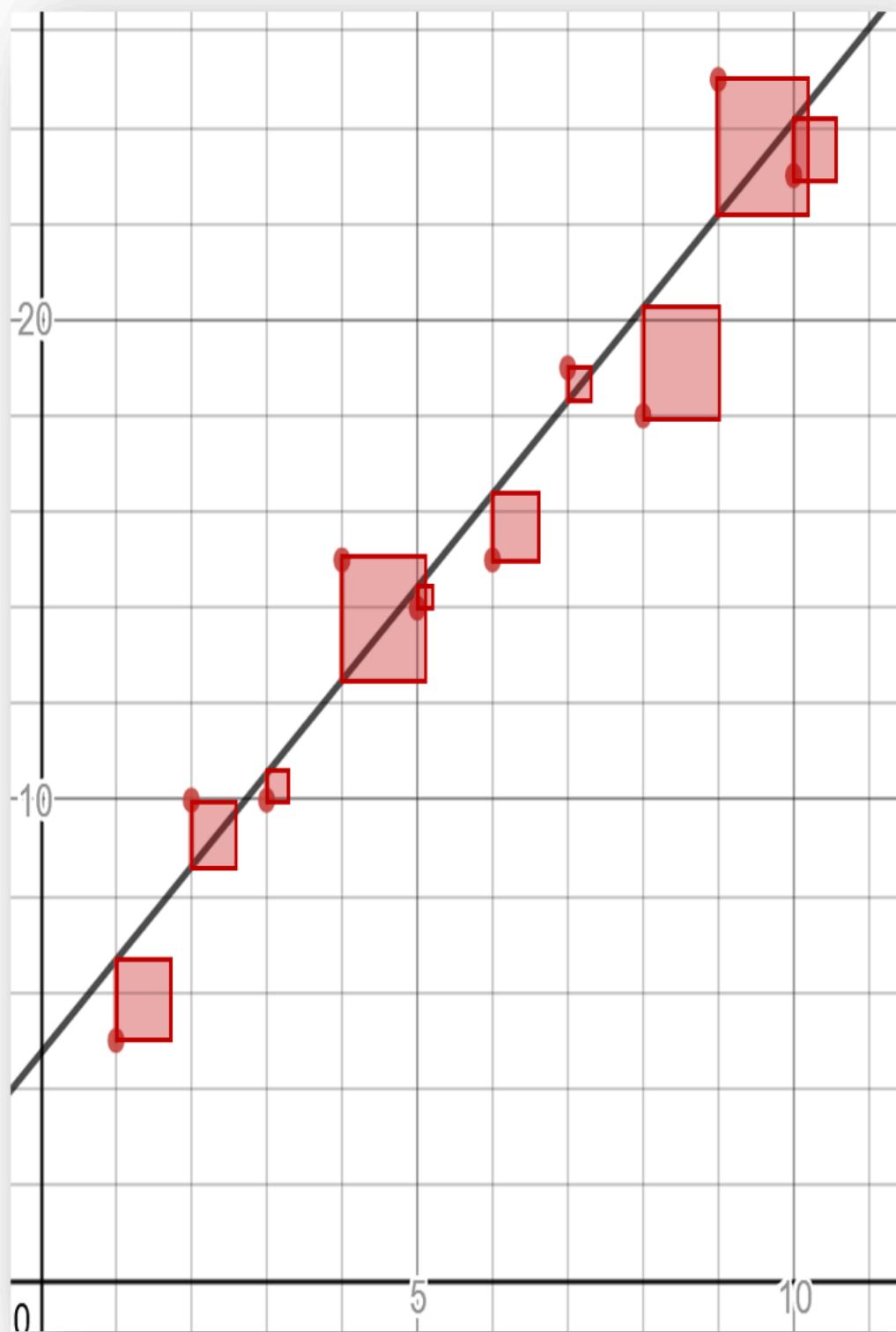


Figure 5-7. Visualizing the sum of squares, which would be the sum of all areas where each square has a side length equal to the residual

Let's modify our code in [Example 5-4](#).

[Example 5-4. Calculating the sum of squares for a given line and data](#)

```
import pandas as pd

# Import points
points = pd.read_csv("https://bit.ly/2KF29Bd").itertuples()

# Test with a given line
m = 1.93939
b = 4.73333

sum_of_squares = 0.0

# calculate sum of squares
for p in points:
    y_actual = p.y
    y_predict = m*p.x + b
    residual_squared = (y_predict - y_actual)**2
    sum_of_squares += residual_squared

print("sum of squares = {}".format(sum_of_squares))
# sum of squares = 28.096969704500005
```

Next question: how do we find the m and b values that will produce the minimum sum of squares, without using a library like Scikit-Learn? Let's look at that next.

Finding the Best Fit Line

We now have a way to measure the quality of a given line against the data points: the sum of squares. The lower we can make that number the better the fit. Now how do we find the right m and b values that creates the *least* sum of squares?

There are a couple of search algorithms we can employ, which try to find the right set of values to solve a given problem. You can try to take a **brute force** approach, generating random m and b values thousands or millions of times

and choosing the ones that produce the least sum of squares. This will not work well because it will take an endless amount of time to find even a decent approximation. We will need something a little more guided. I will curate three techniques you can use: closed form, gradient descent, and stochastic gradient descent. There are other search algorithms like hill climbing that could be used, but we will stick with what's common.

MACHINE LEARNING TRAINING IS FITTING A REGRESSION

This is the heart of “training” a machine learning algorithm. We provide some data and an objective function (the sum of squares) and it will find the right coefficients m and b to fulfill that objective. Let’s explore that next.

Closed Form Equation

Some readers may ask if there is a formula (called a **closed form equation**) to fit a linear regression by exact calculation. The answer is yes, but only for a simple linear regression with one input variable. This luxury does not exist for many machine learning problems with several input variables and a large amount of data that computers cannot handle. We can resort to linear algebra techniques to scale up, and we will talk about this shortly. But at a certain point you will have to resort to search algorithms like stochastic gradient descent, and is why these techniques get emphasis later in the chapter.

For a simple linear regression with only one input and one output variable, here are the closed form equations to calculate m and b . [Example 5-5](#) shows how you can do these calculations in Python.

$$m = \frac{n \sum xy - \sum x \sum y}{n \sum x^2 - (\sum x)^2}$$

$$b = \frac{\sum y}{n} - m \frac{\sum x}{n}$$

Example 5-5. Calculating m and b for a simple linear regression

```
import pandas as pd

# Load the data
points = list(pd.read_csv('https://bit.ly/2KF29Bd', delimiter=',').itertuples())

n = len(points)

m = (n*sum(p.x*p.y for p in points) - sum(p.x for p in points) *
     sum(p.y for p in points)) / (n*sum(p.x**2 for p in points) -
     sum(p.x for p in points)**2)

b = (sum(p.y for p in points) / n) - m * sum(p.x for p in points) / n

print(m, b)
# 1.93939393939394 4.733333333333325
```

These equations to calculate m and b are derived off calculus and proofs beyond the scope of this book. But you can plug in the number of data points n as well as iterate the x and y values to do the operations above. Going forward, we will learn approaches that are more oriented towards contemporary techniques that cope with larger amounts of data. Closed form equations tend to not scale well.

Inverse Matrix Techniques

Going forward, I am going to replace the coefficients m and b with different names, β_1 and β_0 respectively. This is the convention you will see more often in the wild, so it might be a good time to graduate.

While we dedicated an entire chapter to linear algebra in [Chapter 4](#), applying it can be a bit overwhelming when you are new to math and data science. This is why most examples in this book will use plain Python or Scikit-Learn. However, I will sprinkle in a linear algebra where it makes sense, just to show how linear algebra is useful. If you find this section overwhelming feel free to move on to the rest of the chapter and come back later.

We can use transposed and inverse matrices, which we covered in [Chapter 4](#), to fit a linear regression. Below we calculate a vector of coefficients b given a matrix of input variable values X , and a vector of output variable values y .

Without going down a rabbit hole of Calculus and linear algebra proofs, here is the formula:

$$b = (X^T \cdot X)^{-1} \cdot X^T \cdot y$$

You will notice transpose and inverse operations are performed on the matrix X and combined with matrix multiplication. Here is how we perform the above operation in NumPy in [Example 5-6](#) to get our coefficients m and b .

Example 5-6. Using inverse and transposed matrices to fit a linear regression

```
import matplotlib.pyplot as plt
import pandas as pd
from numpy.linalg import inv
import numpy as np

# Import points
df = pd.read_csv('https://bit.ly/3go0Ant', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1].flatten()

# Add placeholder "1" column to generate intercept
X_1 = np.vstack([X, np.ones(len(X))]).T

# Extract output column (all rows, last column)
Y = df.values[:, -1]

# Calculate coefficients for slope and intercept
b = inv(X_1.transpose() @ X_1) @ (X_1.transpose() @ Y)
print(b) # [1.93939394 4.73333333]

# Predict against the y-values
y_predict = X_1.dot(b)
```

It is not intuitive, but note above we have to stack a “column” of 1’s next to our x column. The reason for this is this will generate the intercept β_0 coefficient. Since this column is all 1’s it effectively generates the intercept rather than just a slope β_1 .

When you have a lot of data with a lot of dimensions, computers can start to choke and produce unstable results. This is a use case for matrix

decomposition which we learned about in [Chapter 4](#) on linear algebra. In this specific case, we take our matrix X , append an additional column of 1's to generate the intercept β_0 just like before, and then decompose it into two component matrices Q and R .

$$X = Q \cdot R$$

$$X = Q \cdot R$$

Avoiding more Calculus rabbit holes, here is how we use Q and R to find the beta coefficient values in the matrix form b .

$$b = R^{-1} \cdot Q^T \cdot y$$

And here is how we use the QR decomposition formula above in Python using NumPy to perform a linear regression.

Example 5-7. Using QR decomposition to perform a linear regression

```
import matplotlib.pyplot as plt
import pandas as pd
from numpy.linalg import qr, inv
import numpy as np

# Import points
df = pd.read_csv('https://bit.ly/3go0Ant', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1].flatten()

# Add placeholder "1" column to generate intercept
X_1 = np.vstack([X, np.ones(len(X))]).transpose()

# Extract output column (all rows, last column)
Y = df.values[:, -1]

# calculate coefficients for slope and intercept
# using QR decomposition
Q, R = qr(X_1)
b = inv(R).dot(Q.transpose()).dot(Y)

print(b) # [1.93939394 4.73333333]
```

Typically, QR decomposition is the method used by many scientific libraries for linear regression because it copes with large amounts of data more easily and is more stable.

OVERWHELMED?

If you find these linear algebra examples of linear regression overwhelming, do not worry! I just wanted to provide exposure to a practical use case for linear algebra. Going forward we will focus on other techniques you can utilize.

Gradient Descent

Gradient descent is an optimization technique that uses derivatives and iterations to minimize/maximize a set of parameters against an objective. To learn about gradient descent, let's do a quick thought experiment and then apply it on a simple example.

A Thought Experiment on Gradient Descent

Let's do a thought experiment. Imagine you are in a mountain range at night and given a flashlight. You are trying to get to the lowest point of the mountain range. You can see the slope around you before you even take a step. You step in directions where the slope visibly goes downwards. You take bigger steps for bigger slopes, and smaller steps for smaller slopes. Ultimately you will find yourself at a low point where the slope is flat, a value of 0. Sounds pretty good, right? This approach with the flashlight is known as **gradient descent**, where we step in directions where the slope goes downwards.

In machine learning, we often think of all possible sum of square losses we will encounter with different parameters as a mountainous landscape. We want to minimize our loss and we navigate the loss landscape to do it. To solve this problem gradient descent has an attractive feature: the partial derivative is that flashlight, allowing us to see the slopes for every parameter (in this case m and b , or β_0 and β_1). We step in directions for m and b where the slope goes downwards. We take bigger steps for bigger slopes and

smaller steps for smaller slopes. We can simply calculate the length of this step by taking a fraction of the slope. This fraction is known as our **learning rate**. The higher the learning rate, the faster it will run at the cost of accuracy. But the lower the learning rate, the slower it will take to train and require more iterations.

Let's Walk Before We Run

For the function $f(x) = (x - 3)^2 + 4$, find the x value that produces the lowest point of that function. While we could solve this algebraically, let's use gradient descent to do it.

Here is visually what we are trying to do. As shown in [Figure 5-8](#), we want to “step” x towards the minimum where the slope is 0.

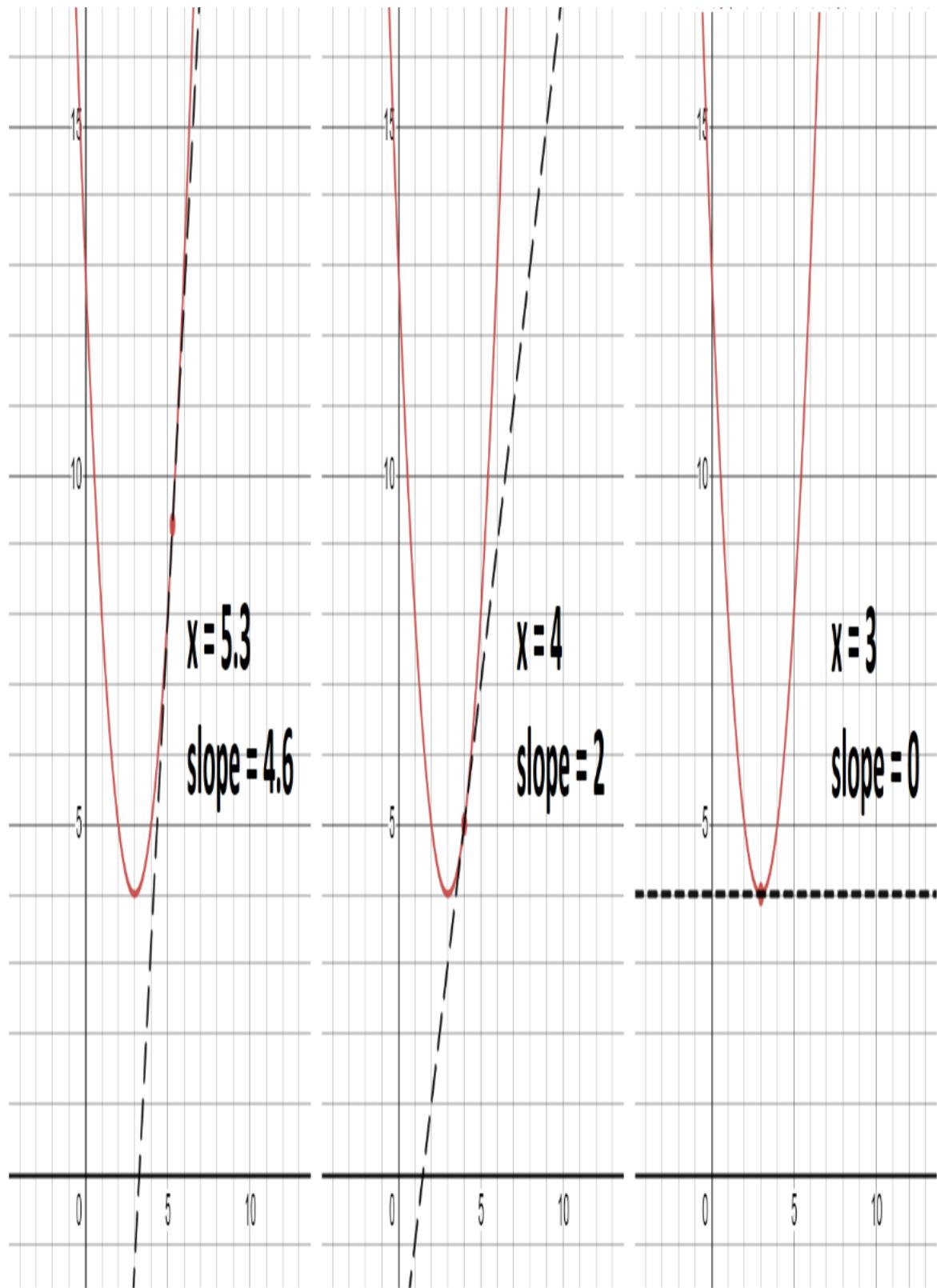


Figure 5-8. Stepping towards the local minimum where the slope approaches 0

In [Example 5-8](#), the function $f(x)$ and its derivative with respect to x is $dx_f(x)$. Recall we covered in [Chapter 1](#) how to use SymPy to calculate derivatives. After finding the derivative, we then proceed to perform gradient descent.

Example 5-8. Using gradient descent to find the minimum of a parabola

```
import random
```

```
def f(x):
    return (x - 3) ** 2 + 4

def dx_f(x):
    return 2*(x - 3)

# The learning rate
L = 0.001

# The number of iterations to perform gradient descent
iterations = 100_000

# start at a random x
x = random.randint(-15,15)

for i in range(iterations):

    # get slope
    d_x = dx_f(x)

    # update x by subtracting the (learning rate) * (slope)
    x -= L * d_x

print(x, f(x)) # prints 2.99999999999889 4.0
```

If we graph the function (as shown in [Figure 5-8](#)) we should see the lowest point of the function is clearly where $x = 3$, and the code above should get very close to that. The learning rate is used to take a fraction of the slope and subtract it from the x value on each iteration. Bigger slopes will result in bigger steps, and smaller slopes will result in smaller steps. After enough iterations, x will end up at the lowest point of the function (or close enough to it) where the slope is 0.

Gradient Descent and Linear Regression

You now might be wondering how do we use this for linear regression. Well it's the same idea except our "variables" are m and b (or β_0 and β_1) rather than x . Here's why: in a simple linear regression already know the x and y values because those are provided as the training data. The "variables" we need to solve are actually the parameters m and b , so we can find the best fit line that will then accept an x variable to predict a new y value.

How do we calculate the slopes for m and b ? We need the partial derivatives for each of these. What function are we taking the derivative of? Remember we are trying to minimize loss and that will be the sum of squares. So we need to find the derivatives of our sum of squares function with respect to m and b .

I implement these two partial derivatives for m and b as shown in **Example 5-9**. We will learn how to do this shortly in SymPy. I then perform gradient descent to find m and b . 100,000 iterations with a learning rate of .001 will be sufficient. Note that the smaller you make that learning rate, the slower and more iterations you will need. But if you make it too high it will run fast but have a poor approximation. When someone says a machine learning algorithm is "learning" or "training", it really is just fitting a regression like this.

Example 5-9. Performing gradient descent for a linear regression

```
import pandas as pd

# Import points from CSV
points = list(pd.read_csv("https://bit.ly/2KF29Bd").itertuples())

# Building the model
m = 0.0
b = 0.0

# The learning Rate
L = .001

# The number of iterations
iterations = 100_000

n = float(len(points)) # Number of elements in X

# Perform Gradient Descent
```

```

for i in range(iterations):

    # slope with respect to m
    D_m = sum(2 * p.x * ((m * p.x + b) - p.y) for p in points)

    # slope with respect to b
    D_b = sum(2 * ((m * p.x + b) - p.y) for p in points)

    # update m and b
    m -= L * D_m
    b -= L * D_b

print("y = {0}x + {1}".format(m, b))
# y = 1.9393939393939548x + 4.733333333333227

```

Well not bad! That approximation got close to our closed form equation solution. But what's the catch? Just because we found the “best fit line” by minimizing sum of squares does not mean our linear regression is any good. Does minimizing the sum of squares guarantee a great model to make predictions? Not exactly. Now that I showed you how to fit a linear regression let's take a step back and revist the big picture, and determine whether a given linear regression is the right way to make predictions in the first place.

Gradient Descent for Linear Regression Using SymPy

If you want the SymPy code that came up with these two derivatives for the sum of squares function, for m and b respectively, here is the code in [Example 5-10](#).

Example 5-10. Calculating partial derivatives for m and b

```

from sympy import *

m, b, i, n = symbols('m b i n')
x, y = symbols('x y', cls=Function)

sum_of_squares = Sum((m*x(i) + b - y(i)) ** 2, (i, 0, n))

d_m = diff(sum_of_squares, m)
d_b = diff(sum_of_squares, b)
print(d_m)
print(d_b)

```

```
# OUTPUTS
# Sum(2*(b + m*x(i) - y(i))*x(i), (i, 0, n))
# Sum(2*b + 2*m*x(i) - 2*y(i), (i, 0, n))
```

You will see above the two derivatives for m and b respectively printed. Note the `Sum()` function will iterate and add items together (in this case all the data points), and we treat x and y as functions that look up a value for a given point at index i .

In mathematical notation, where $e(x)$ represents the sum of squares loss function, here are the partial derivatives for m and b .

$$e(x) = \sum_{i=0}^n ((mx_i + b) - y_i)$$

$$\frac{d}{dm} e(x) = \sum_{i=0}^n 2(b + mx_i - y_i)x_i$$

$$\frac{d}{db} e(x) = \sum_{i=0}^n (2b + 2mx_i - 2y_i)$$

If you want to apply our dataset and execute a linear regression using gradient descent, you will have to perform a few additional steps as shown in [Example 5-11](#). We will need to substitute for the n , $x(i)$ and $y(i)$ values iterating all of our data points for the `d_m` and `d_b` derivative functions. That should leave only the m and b variables which we will search for the optimal values using gradient descent.

Example 5-11.

```
import pandas as pd
from sympy import *

# Import points from CSV
points = list(pd.read_csv("https://bit.ly/2KF29Bd").itertuples())

m, b, i, n = symbols('m b i n')
x, y = symbols('x y', cls=Function)

sum_of_squares = Sum((m*x(i) + b - y(i)) ** 2, (i, 0, n))
```

```

d_m = diff(sum_of_squares, m) \
    .subs(n, len(points) - 1).doit() \
    .replace(x, lambda i: points[i].x) \
    .replace(y, lambda i: points[i].y)

d_b = diff(sum_of_squares, b) \
    .subs(n, len(points) - 1).doit() \
    .replace(x, lambda i: points[i].x) \
    .replace(y, lambda i: points[i].y)

# compile using lambdify for faster computation
d_m = lambdify([m, b], d_m)
d_b = lambdify([m, b], d_b)

# Building the model
m = 0.0
b = 0.0

# The learning Rate
L = .001

# The number of iterations
iterations = 100_000

# Perform Gradient Descent
for i in range(iterations):

    # update m and b
    m -= d_m(m,b) * L
    b -= d_b(m,b) * L

print("y = {0}x + {1}".format(m, b))
# y = 1.9393939393954x + 4.7333333333231

```

As shown in [Example 5-11](#), it is a good idea to call `lambdify()` on both of our partial derivative functions to convert them from SymPy to an optimized Python function. This will cause computations to perform much more quickly when we do gradient descent. The resulting Python functions are backed by NumPy, SciPy, or whatever numerical libraries SymPy detects you have available. After that, we can perform gradient descent.

Overfitting and Variance

Riddle me this: if we truly wanted to minimize loss, as in reduce the sum of squares to 0, what would we do? Are there other options than linear regression? One conclusion you may arrive at is simply fit a curve that touches all the points. Heck, why not just connect the points together in segments and use that to make predictions as shown in [Figure 5-9](#)? That gives us a loss of 0!

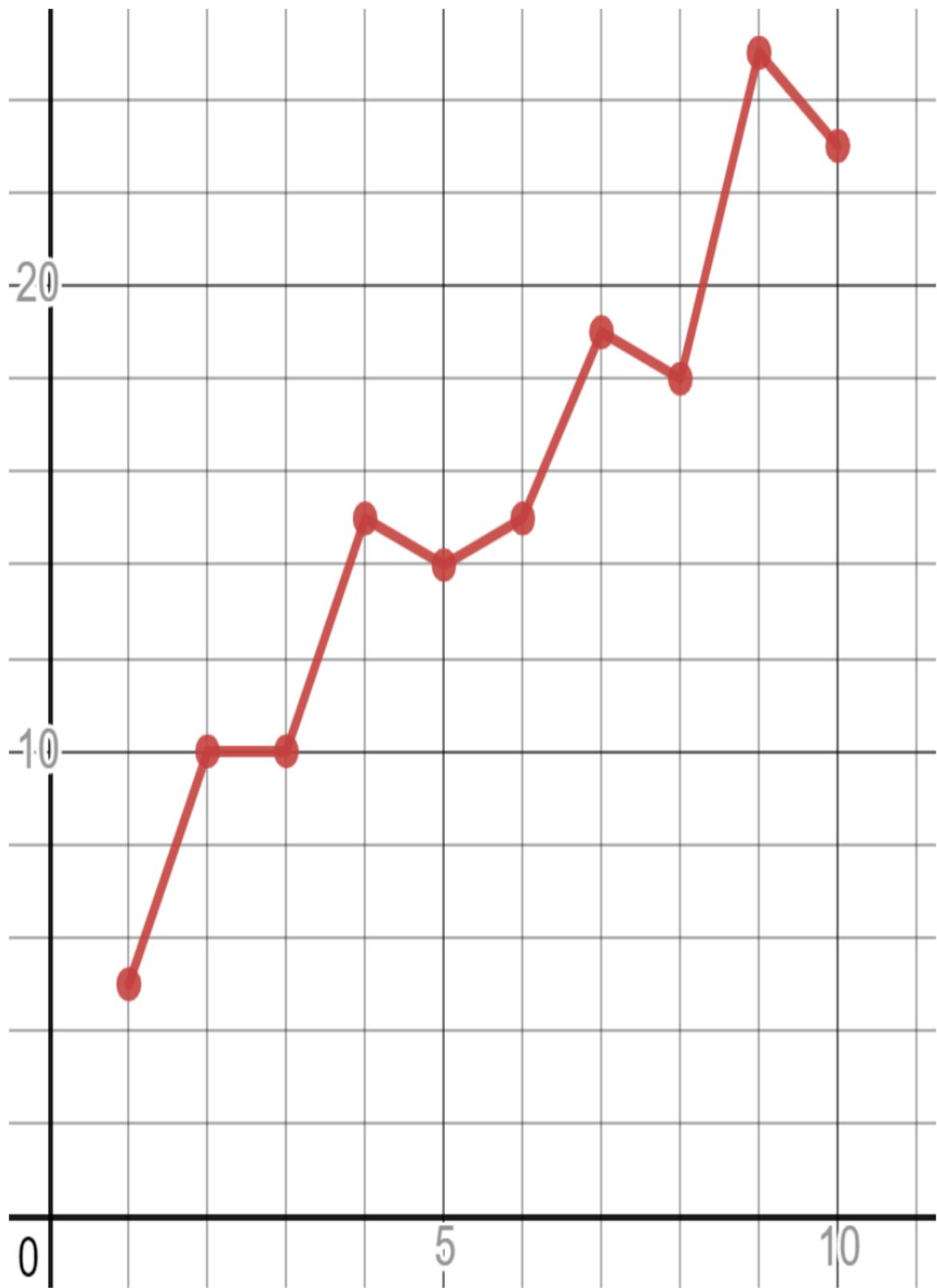


Figure 5-9. Performing a regression by simply connecting the points, resulting in zero loss.

Shoot, why did we go through all that trouble with linear regression and not do this instead? Well, remember our big picture objective is not to minimize the sum of squares but to make accurate predictions on new data. This “connect the dots” model is severely **overfit**, meaning it shaped the regression to the training data too exactly to the point it will predict poorly on new data. This simple “connect the dots” model is sensitive to outliers that are far away from the rest of the points, meaning it will have high **variance** in predictions. Because overfitting increases variance, predictions are going to be all over the place!

OVERFITTING IS MEMORIZATION

When you hear someone say a regression “memorized” the data rather than generalizing it, they are talking about overfitting.

As you can guess we want to find effective generalizations in our model rather than memorize data. Otherwise our regression simply turns into a database where we look up values.

This is why in machine learning you will find bias is added to the model, and linear regression is considered a highly biased model. This is not the same as bias in the data, which we talked about extensively in [Chapter 3](#). **Bias in a model** means we prioritize a method (e.g. maintaining a straight line) as opposed to bending and fitting to exactly what the data says. A biased model leaves some wiggle room hoping to minimize loss on new data for better predictions, as opposed to minimizing loss on data it was trained on. I guess you could say adding bias to a model counteracts overfitting with **underfitting**, or fitting less to the training data.

As you can imagine this is a balancing act because it is two contradictory objectives. In machine learning, we basically are saying “I want to fit a regression to my data, but I don’t want to fit it *too much*. I need some wiggle room for predictions on new data that will be different.”

Still we cannot just apply a linear regression to some data, make some predictions with it, and assume all is okay. A linear regression can still

overfit even with the bias of a straight line. Therefore, we need to check and mitigate for both overfitting and underfitting to find the sweet spot between the two. That is if there is one at all, in which case you should abandon the model altogether. Don't p-hack, people!

Stochastic Gradient Descent

In a machine learning context, you are unlikely to do gradient descent in practice like we did earlier, where we trained on all training data (called **batch gradient descent**). In practice you are more likely to perform **stochastic gradient descent**, which will only train on one sample of the dataset on each iteration. In **mini-batch gradient descent**, multiple samples of the dataset are used (e.g. 10 or 100 data points) on each iteration.

Why only use part of the data on each iteration? Machine learning practitioners cite a few benefits. First it reduces computation significantly, as each iteration does not have to traverse the entire training dataset but only part of it. The second benefit is it reduces overfitting. By exposing the training algorithm to only part of the data on each iteration, it keeps changing the loss landscape so it does not settle in the loss minimum. After all, minimizing the loss is what causes overfitting and so we introduce some randomness to create underfitting.

Of course, our approximation becomes loose so we have to be careful. This is why we will talk about training/testing splits shortly, as well as other metrics to evaluate our linear regression's reliability.

Example 5-12 shows how to perform stochastic gradient descent in Python. If you change the sample size to be more than 1, it will perform mini-batch gradient descent.

Example 5-12. Performing stochastic gradient descent for a linear regression

```
import pandas as pd  
import numpy as np
```

```
# Input data
```

```

data = pd.read_csv('https://bit.ly/2KF29Bd', header=0)

X = data.iloc[:, 0].values
Y = data.iloc[:, 1].values

n = data.shape[0] # rows

# Building the model
m = 0.0
b = 0.0

sample_size = 1 # sample size
L = .0001 # The learning Rate
epochs = 1_000_000 # The number of iterations to perform gradient descent

# Performing Stochastic Gradient Descent
for i in range(epochs):
    idx = np.random.choice(n, sample_size, replace=False)
    x_sample = X[idx]
    y_sample = Y[idx]

    # The current predicted value of Y
    Y_pred = m * x_sample + b

    # d/dm derivative of loss function
    D_m = (-2 / sample_size) * sum(x_sample * (y_sample - Y_pred))

    # d/db derivative of loss function
    D_b = (-2 / sample_size) * sum(y_sample - Y_pred)

    m = m - L * D_m # Update m
    b = b - L * D_b # Update b

    # print progress
    if i % 10000 == 0:
        print(i, m, b)

print("y = {}x + {}".format(m, b))

```

When I ran this, I got a linear regression of $y = 1.9382830354181135x + 4.753408787648379$. Obviously your results are going to be different, and because of stochastic gradient descent we really aren't going to converge towards a specific minimum but end up in a broader neighborhood of coefficient values.

IS RANDOMNESS BAD?

If this randomness feels uncomfortable where you get a different answer every time you run a piece of code, welcome to the world of machine learning, optimization, and stochastic algorithms! Many algorithms that do approximations are random-based, and while some are extremely useful others can be sloppy and perform poorly as you might expect.

A lot of people look to machine learning and AI as some tool that gives objective and precise answers, but that cannot be further from the truth. Machine learning can be misused as a cursory form of statistics that approximates (sometimes very liberally) based on a combination of data and random-based search strategies.

While randomness can create some powerful tools, it can also be abused. Be careful to not use seed values and randomness to p-hack a “good” result, and put effort into analyzing your data and model if it’s possible.

The Correlation Coefficient

Take a look at this scatterplot in [Figure 5-10](#) alongside its linear regression. Why might a linear regression not work too well here?

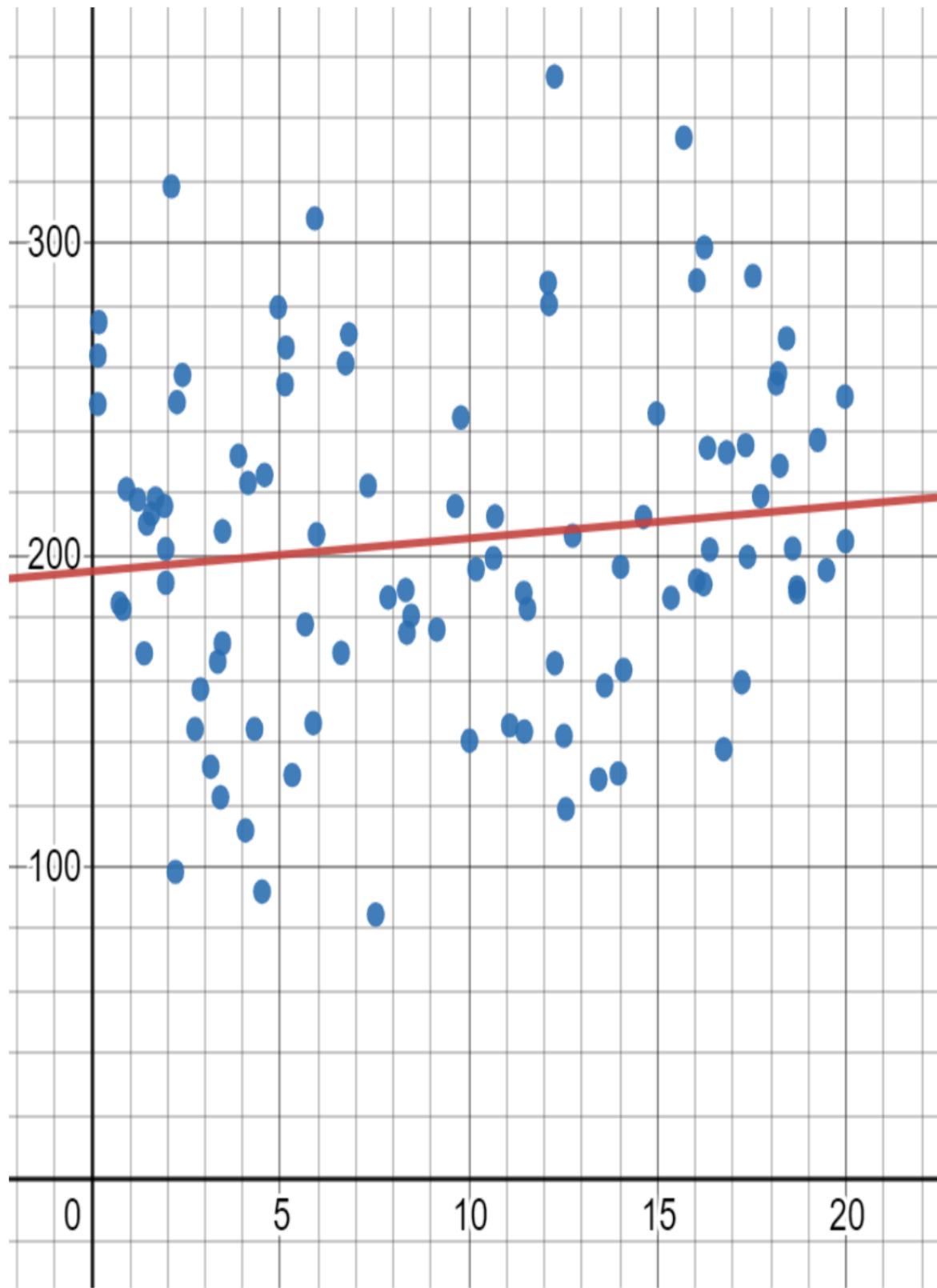


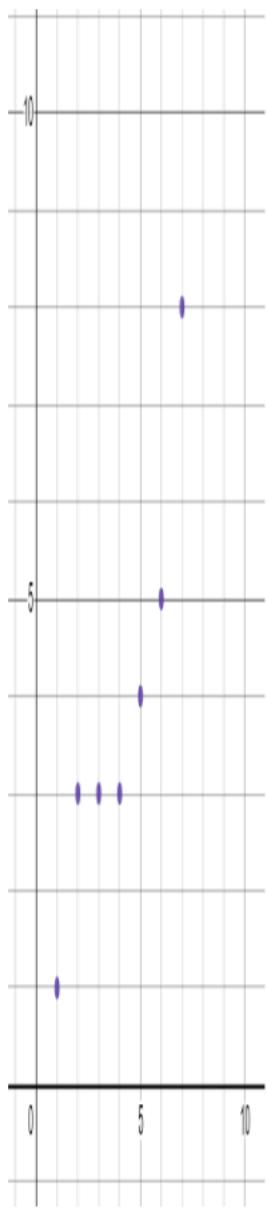
Figure 5-10. A scatterplot of data with high variance

The problem here is that the data has high variance. If the data is extremely spread out, it is going to drive up the variance to the point predictions become less accurate and useful, resulting in large residuals. Of course we can introduce a biased model, such as linear regression, to not bend and respond to the variance so easily. However the underfitting is also going to undermine our predictions because the data is so spread out. We need to numerically measure how “off” our predictions are.

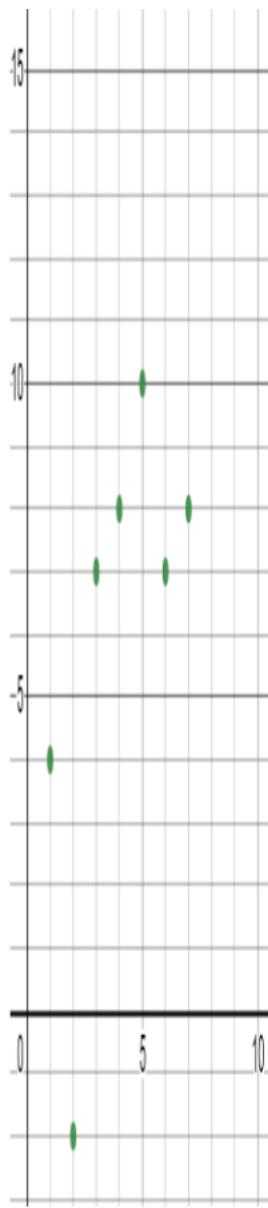
So how do you measure these residuals in aggregate? How do you also get a sense for how bad the variance in the data is? Let me introduce you to the **correlation coefficient**, also called the **Pearson correlation**, which measures the strength of the relationship between two variables as a value between -1 and 1. A correlation coefficient closer to 0 indicates there is no correlation. A correlation coefficient closer to 1 indicates a strong **positive correlation**, meaning when one variable increases the other proportionally increases. If it is closer to -1 then it indicates a strong **negative correlation**, which means as one variable increases the other proportionally decreases.

Note the correlation coefficient is often denoted as r . The highly scattered data in [Figure 5-15](#) has a correlation coefficient of 0.1201. Since it is much closer to 0 than 1, we can infer it has little correlation.

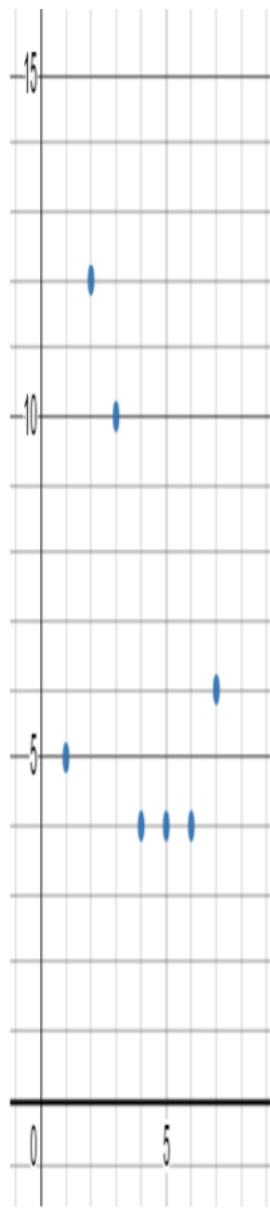
Here are three other scatterplots in [Figure 5-11](#) showing their correlation coefficients. Notice that the more the points follow a line, the stronger the correlation. More dispersed points result in weaker correlations.



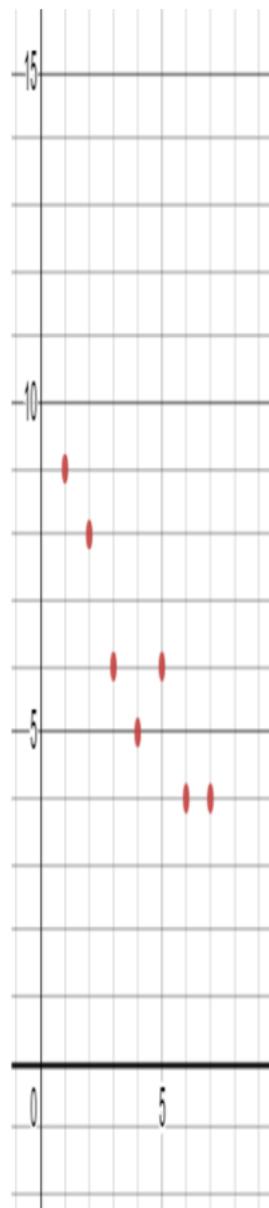
PEARSON CORRELATION: .923133



PEARSON CORRELATION: .643237



PEARSON CORRELATION: -.44984



PEARSON CORRELATION: .9267

Figure 5-11. Correlation coefficients for four scatterplots

As you can imagine, the correlation coefficient is useful to see if there is a possible relationship between two variables. If there is a strong positive/negative relationship, it will be useful in our linear regression. If there is not a relationship, they may just add noise and hurt model accuracy.

How do we use Python to calculate the correlation coefficient? Let's use our simple **10-point dataset** we used earlier. A quick and easy way to analyze correlations for all pairs of variables is using Pandas' `corr()` function. It makes it easy to see the correlation coefficient between every pair of variables in a dataset, which in this case will only be x and y . This is known as a **correlation matrix**. Take a look in **Example 5-13**.

Example 5-13. Using Pandas to see the correlation coefficient between every pair of variables

```
import pandas as pd

# Read data into Pandas dataframe
df = pd.read_csv('https://bit.ly/2KF29Bd', delimiter=',')

# Print correlations between variables
correlations = df.corr(method='pearson')
print(correlations)

# OUTPUT:
#          x      y
# x  1.000000  0.957586
# y  0.957586  1.000000
```

As you can see the correlation coefficient .957586 between x and y indicates a strong positive correlation between the two variables. You can ignore the parts of the matrix where x or y are set to themselves and have a value of 1.0. Obviously when x or y is set to itself the correlation will be perfect at 1.0, because their values match themselves exactly. When you have more than two variables, the correlation matrix will show a larger grid because there are more variables to pair and compare.

If you change the code to use a different dataset with a lot of variance, where the data is spread out, you will see that correlation coefficient decreases.

This again indicates a weaker correlation.

CALCULATING THE CORRELATION COEFFICIENT

For those that are mathematically curious how the correlation coefficient is calculated, here is the formula:

$$r = \frac{n \sum xy - (\sum x)(\sum y)}{\sqrt{n \sum x^2 - (\sum x^2)} \sqrt{n \sum y^2 - (\sum y^2)}}$$

If you want to see this fully implemented in Python, I like to use one-line `for` loops to do those summations. **Example 5-14** shows a correlation coefficient implemented from scratch in Python.

Example 5-14. Calculating correlation coefficient from scratch in Python

```
import pandas as pd
from math import sqrt

# Import points from CSV
points = list(pd.read_csv("https://bit.ly/2KF29Bd").itertuples())
n = len(points)

numerator = n * sum(p.x * p.y for p in points) - \
            sum(p.x for p in points) * sum(p.y for p in points)

denominator = sqrt(n*sum(p.x**2 for p in points) - sum(p.x for p in points)**2) \
              * sqrt(n*sum(p.y**2 for p in points) - sum(p.y for p in points)**2)

corr = numerator / denominator

print(corr)

# OUTPUT:
# 0.9575860952087218
```

Statistical Significance

Here is another aspect to a linear regression you must consider: is my data correlation coincidental? In Chapter 3 we studied hypothesis testing and p-values, and we are going to extend those ideas here with a linear regression.

THE STATSMODEL LIBRARY

While we are not going to introduce yet another library in this book, it is worth mentioning that `statsmodel` is noteworthy if you want to do statistical analysis.

<https://www.statsmodels.org/stable/regression.html>

Scikit-Learn and other machine learning libraries do not provide tools for statistical significance and confidence intervals for reasons we will discuss in another sidebar. We are going to code those techniques ourselves. But know this library exists and it is worth checking out!

Let's start with a fundamental question: is it possible I see a linear relationship in my data due to random chance? How can we be 95% sure the correlation between these two variables is real, not coincidental? If this sounds like a hypothesis test from Chapter 3, it's because it is! We need to express not just the correlation coefficient, but quantify how confident we are that the correlation coefficient did not occur by chance.

Rather than estimating a mean like we did in [Chapter 3](#) with the drug testing example, we are estimating the population correlation coefficient based on a sample. We denote the population correlation coefficient with the Greek symbol ρ ("Rho") while our sample correlation coefficient is r . Just like we did in Chapter 3, we will have a null hypothesis H_0 and alternative hypothesis H_1 .

$$H_0 : \rho = 0 \text{ (implies no relationship)}$$

$$H_1 : \rho \neq 0 \text{ (relationship is present)}$$

Our null hypothesis H_0 is that there is no relationship between two variables, or more technically the correlation coefficient is 0. The alternative

hypothesis H_1 is there is a relationship, and it can be a positive or negative correlation. This is why the alternative hypothesis is defined as $\rho \neq 0$ to support both a positive and negative correlation.

Let's return to our dataset of 10 points as shown in [Figure 5-12](#). How likely is it we would see these data points by chance? And they happen to produce what looks like a linear relationship?

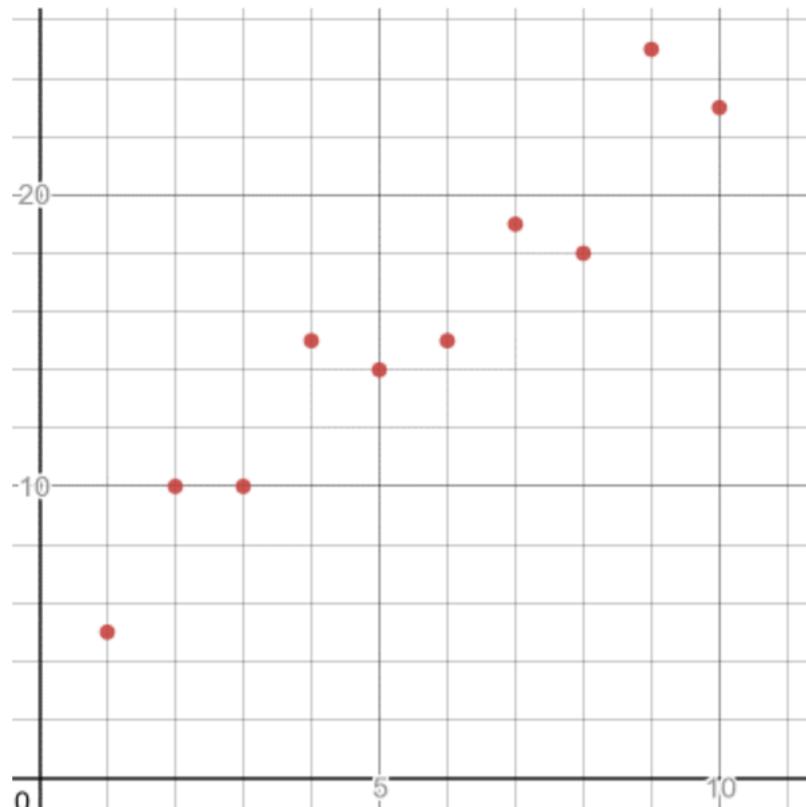


Figure 5-12. How likely would we see this data, which seems to have a linear correlation, by random chance?

We already calculated the correlation coefficient for this dataset earlier in [Example 5-13](#), which is 0.957586. That's a strong and compelling positive correlation. But again we need to evaluate if this was by random luck. Let's pursue our hypothesis test with 95% confidence using a two-tailed test, exploring if there is a relationship between these two variables.

We talked about the T-distribution in [Chapter 3](#), which has fatter tails to capture more variance and uncertainty. We use a T-distribution rather than a normal distribution to do hypothesis testing with linear regression. First, let's

plot a T-distribution with a 95% critical value range as shown in [Figure 5-13](#). We account for the fact there are 10 records in our sample and therefore we have 9 degrees of freedom ($10 - 1 = 9$).

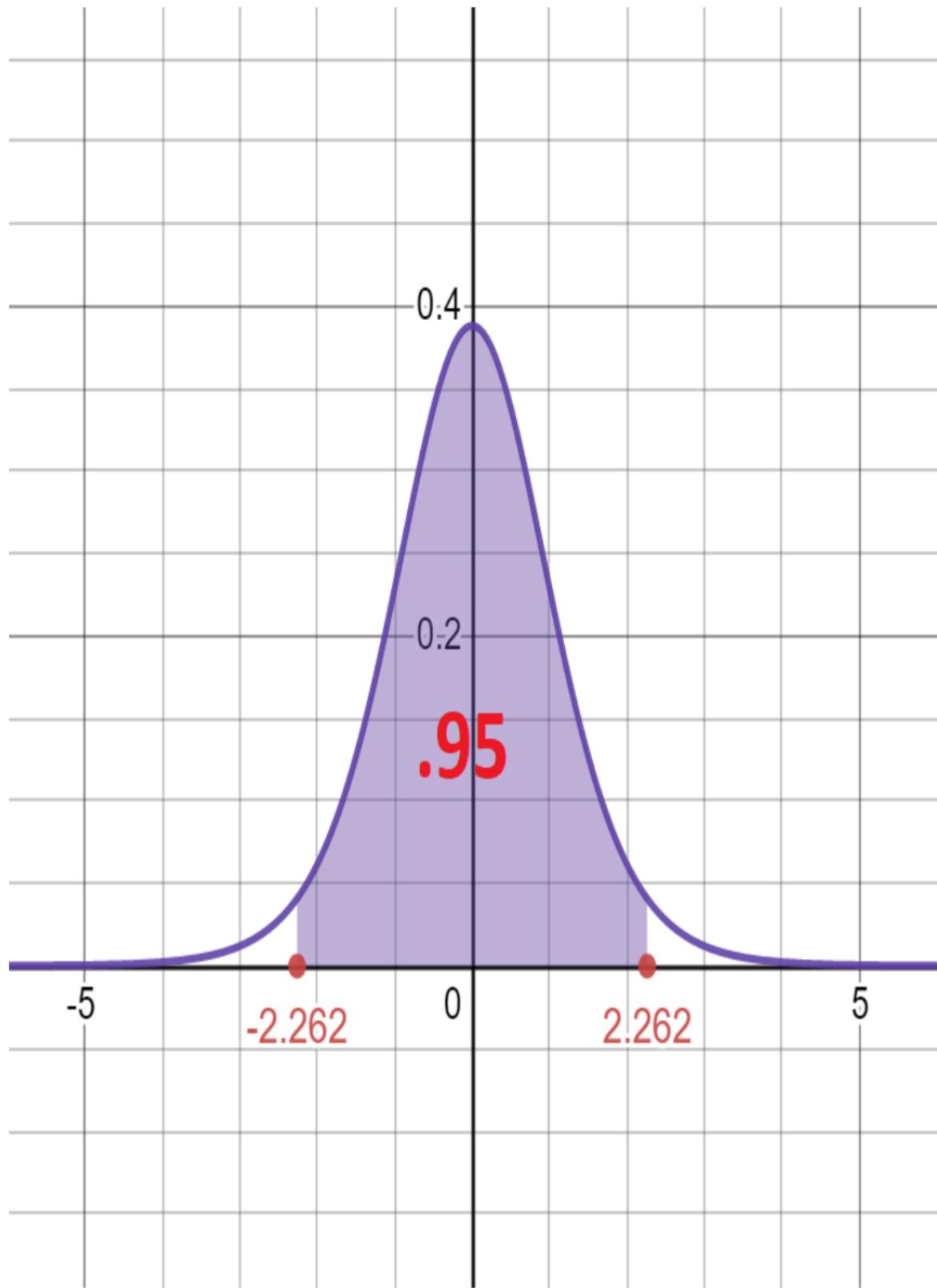


Figure 5-13. A T-Distribution with 9 degrees of freedom, as there are 10 records and we subtract 1.

The critical value is approximately ± 2.262 , and we can calculate that in Python as shown in [Example 5-15](#). This captures 95% of the center area of our T-distribution.

Example 5-15. Calculating the critical value from a T-distribution

```
from scipy.stats import t

n = 10
lower_cv = t(n-1).ppf(.025)
upper_cv = t(n-1).ppf(.975)

print(lower_cv, upper_cv)
# -2.262157162740992 2.2621571627409915
```

If our test value happens to fall outside this range of $(-2.262, 2.262)$ then we can reject our null hypothesis and say our correlation is real (not coincidental) with at least 95% confidence. To calculate the test value t , we need to use this formula below. Again r is the correlation coefficient and n is the sample size.

$$t = \frac{r}{\sqrt{\frac{1-r^2}{n-2}}}$$
$$t = \frac{.957586}{\sqrt{\frac{1-.957586^2}{10-2}}} = 9.39956$$

Let's put the whole test together in Python as shown in [Example 5-16](#). If our test value falls outside the critical range of 95% confidence, we accept that our correlation was not by chance.

Example 5-16. Testing significance for linear-looking data

```
from scipy.stats import t
from math import sqrt

# sample size
n = 10

lower_cv = t(n-1).ppf(.025)
upper_cv = t(n-1).ppf(.975)
```

```

# correlation coefficient
# derived from data https://bit.ly/2KF29Bd
r = 0.957586

# Perform the test
test_value = r / sqrt((1-r**2) / (n-2))

print("TEST VALUE: {}".format(test_value))
print("CRITICAL RANGE: {}, {}".format(lower_cv, upper_cv))

if test_value < lower_cv or test_value > upper_cv:
    print("CORRELATION PROVEN, REJECT H0")
else:
    print("CORRELATION NOT PROVEN, FAILED TO REJECT H0 ")

# Calculate p-value
if test_value > 0:
    p_value = 1.0 - t(n-1).cdf(test_value)
else:
    p_value = t(n-1).cdf(test_value)

# Two-tailed, so multiply by 2
p_value = p_value * 2
print("P-VALUE: {}".format(p_value))

```

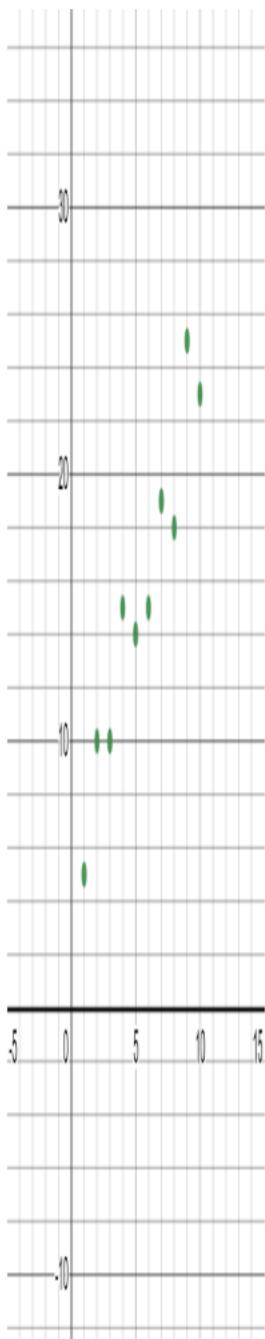
The test value above is approximately 9.39956, which is definitely outside the range of (-2.262, 2.262) so we can reject the null hypothesis and say our correlation is real. That's because the p-value is remarkably significant: .000005976. This is well below our .05 threshold, so this is virtually not coincidence there is a correlation. I'll confess I actually fabricated this dataset, and it makes sense the p-value is so small because the points strongly resemble a line. It is highly unlikely these points randomly arranged themselves near a line this closely by chance.

Figure 5-14 shows some other datasets with their correlation coefficients and p-values. Analyze each one of them. Which one is likely the most useful for predictions? What are the problems with the other ones?



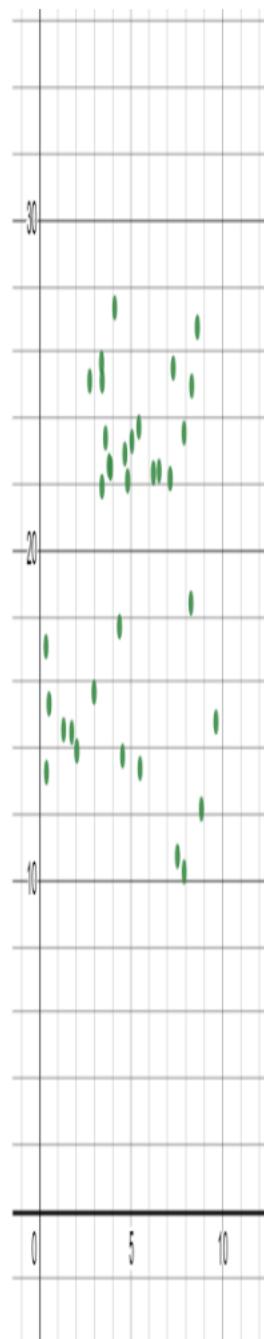
r 0.771454

p-value 0.34913



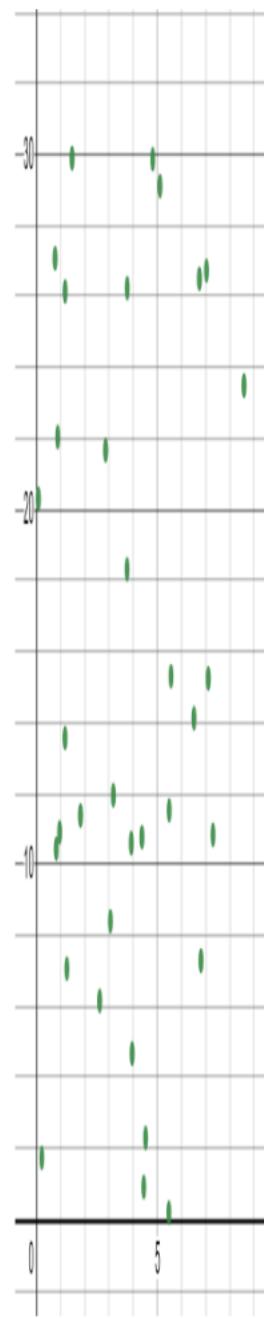
r 0.957586095

p-value 0.000005976



r 0.100984708

p-value 0.569732057



r 0.037875905

p-value 0.831544971

Figure 5-14. Different datasets and their correlation coefficients and p-values

Now that you've had a chance to work through the analysis on the datasets from Figure [Figure 5-14](#), let's walk through the findings. The left figure has a high positive correlation, but it only has 3 points. The lack of data drives up the p-value significantly to .34913 and increases the likelihood the data happened by chance. This makes sense because having just 3 data points makes it likely to see a linear pattern, but it's not much better than having 2 points that will simply connect a line between the two points. This brings up an important rule-of-thumb: having more data will decrease your p-value, especially if that data gravitates towards a line.

The second figure is what we just covered. It only has 10 data points, but it forms a linear pattern so nicely, we not only have a strong positive correlation but also an extremely low p-value. When you have a p-value this low, you can bet you are measuring an engineered and tightly controlled process, not something sociological or natural.

The right two figures in [Figure 5-14](#) perform poorly for a linear model. Their correlation coefficient is close to 0 indicating no correlation, and the p-values unsurprisingly indicate randomness played a role.

The rule of thumb is this: the more data you have that consistently resembles a line, the more significant the p-value for your correlation will be. The more scattered or sparse the data, the more the p-value will increase and thus indicate your correlation occurred by random chance.

Coefficient of Determination

Let's learn an important metric you will see a lot in statistics and machine learning regressions. The **coefficient of determination**, called r^2 , measures how much variation in one variable is explainable by the variation of the other variable. It is also the square of the correlation coefficient r . As r approaches a perfect correlation (-1 or 1), the r^2 approaches 1. Essentially, r^2 shows how much two variables interact with each other.

Let's continue looking at our data from [Figure 5-12](#). In [Example 5-17](#), take our dataframe code from earlier that calculates the correlation coefficient and then simply square it. That will multiply each correlation coefficient by itself.

Example 5-17. Creating a correlation matrix in Pandas

```
import pandas as pd

# Read data into Pandas dataframe
df = pd.read_csv('https://bit.ly/2KF29Bd', delimiter=",")

# Print correlations between variables
coeff_determination = df.corr(method='pearson') ** 2
print(coeff_determination)

# OUTPUT:
#           x      y
# x  1.000000  0.916971
# y  0.916971  1.000000
```

A coefficient of determination of .916971 is interpreted as “91.6971% of the variation in x is explained by y (and vice versa), and the remaining 8.3029% is noise caused by other uncaptured variables.” 0.916971 is a pretty good coefficient of determination, showing that x and y explain each other’s variance. But there could be other variables at play making up that remaining 0.083029. Remember, correlation does not equal causation, so there could be other variables contributing to the relationship we are seeing.

CORRELATION IS NOT CAUSATION!

It is important to note that while we put a lot of emphasis on measuring correlation and building metrics around it, please note *correlation is not causation!* You probably have heard that mantra before but I want to expand on why statisticians say it.

Just because we see a correlation between x and y does not mean x causes y . It could actually be y causes x ! Or maybe there is a third uncaptured variable z that is causing x and y . It could be that x and y do not cause each other at all and the correlation is just coincidental, hence why it is important we measure the statistical significance.

Now I have a more pressing question for you. Can computers discern between correlation and causation? The answer is a resounding “NO!” Computers only have concept of correlation but not causation. Let’s say I load a dataset to Scikit-Learn showing gallons of water consumed and my water bill. My computer, or any program including Scikit-Learn, does not have any notion of whether more water usage causes a higher bill, or a higher bill causes more water usage. An “AI” system could easily conclude the latter, as nonsensical as that is. This is why many machine learning projects require a human in the loop to inject common sense.

In computer vision, this happens too. Computer vision often uses a regression on the numeric pixels to predict a category. If I train a computer vision system to recognize cows using pictures of cows, it can easily make correlations with the field rather than cows. Therefore if I show a picture of an empty field, it will label the grass as cows! This again is because computers have no concept of causality (the cow shape should cause the label “cow”) and instead gets lost in correlations we are not interested in.

Standard Error of the Estimate

One way to measure the overall error of a linear regression is the **SSE, or sum of squared error**. We learned about this earlier where we square each residual and sum them. If \hat{y} (pronounced “y-hat”) is each predicted value from the line and y represents each actual y value from the data, here is the calculation:

$$SSE = \sum (y - \hat{y})^2$$

However, all of these squared values are hard to interpret so we can use some square root logic to scale things back into their original units. We will also average all of them, and this is what the **standard error of the estimate**

(S_e) does. If n is the number of data points, Example 5-18 shows how we calculate the standard error S_e in Python.

$$S_e = \frac{\sum (y - \hat{y})^2}{n - 2}$$

Example 5-18. Calculating the standard error of the estimate

Here is how we calculate it in Python:

```
import pandas as pd
from math import sqrt

# Load the data
points = list(pd.read_csv('https://bit.ly/2KF29Bd', delimiter=',').itertuples())

n = len(points)

# Regression line
m = 1.939
b = 4.733

# Calculate Standard Error of Estimate
s_e = sqrt((sum((p.y - (m*p.x +b))**2 for p in points))/(n-2))

print(s_e)
# 1.87406793500129
```

Why $n - 2$ instead of $n - 1$, like we have done in so many variance calculations in Chapter 3? Without going too deep into mathematical proofs, this is because a linear regression has two variables, not just one, so we have to increase the uncertainty by one more in our degrees of freedom.

You will notice the standard error of the estimate looks remarkably similar to the standard deviation we studied in Chapter 3. This is not by accident. That is because it is the standard deviation for a linear regression.

Prediction Intervals

As mentioned earlier, our data in a linear regression is a sample from a population. Therefore our regression is only as good as our sample. Our

linear regression line also has a normal distribution running along it. Effectively, this makes each predicted y value a sample statistic just like the mean. As a matter of fact, the “mean” is shifting along the line.

Remember when we talked statistics in [Chapter 2](#) about variance and standard deviation? The concepts apply here too. With a linear regression, we hope that data follows a normal distribution in a linear fashion. A regression line serves as the shifting “mean” of our bell curve and the spread of the data around the line reflects the variance/standard deviation, as shown in [Figure 5-15](#).

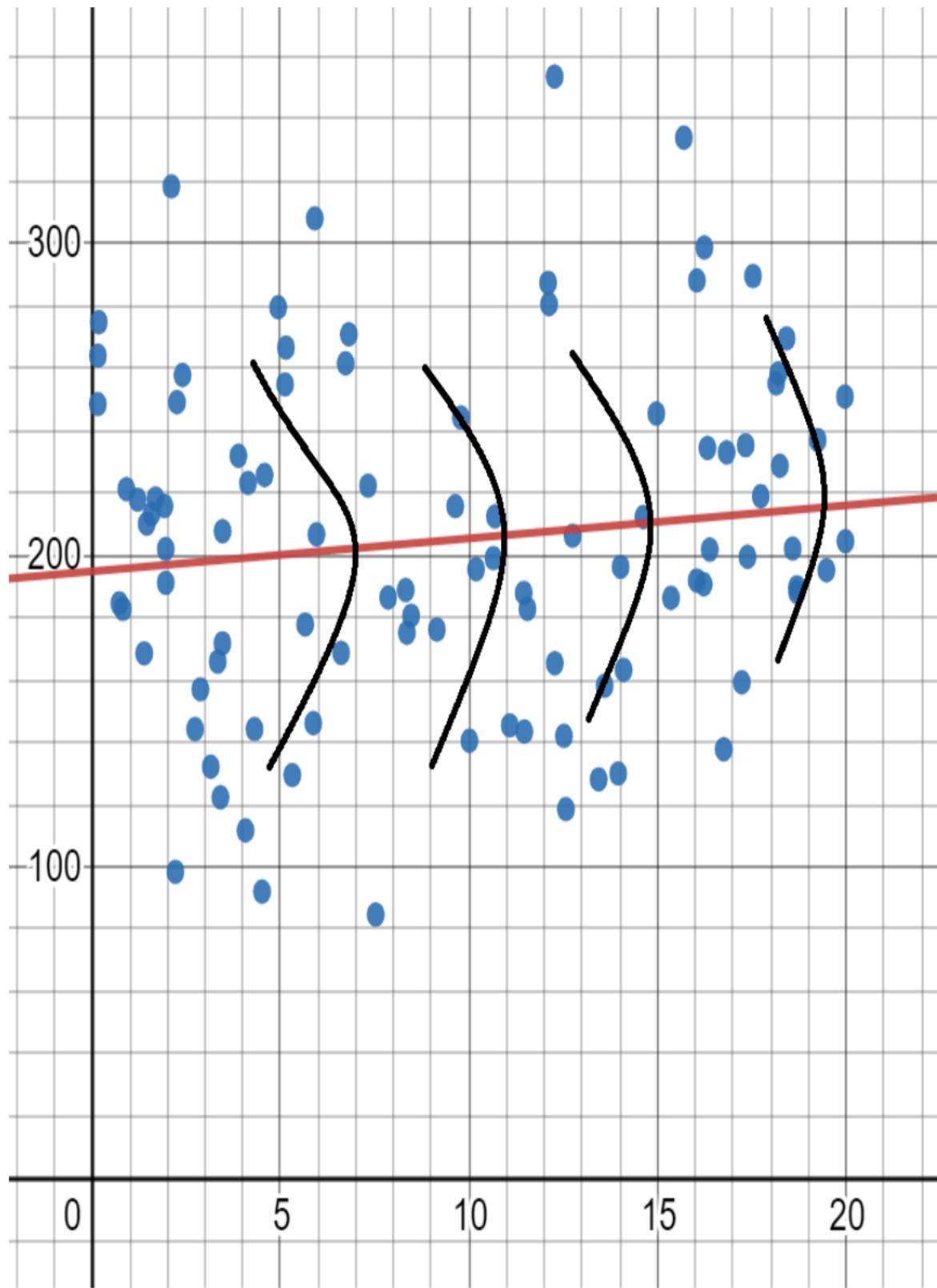


Figure 5-15. A linear regression assumes a normal distribution is following the line

When we have a normal distribution following a linear regression line, it is known as a **bivariate normal distribution**. This makes sense because we have not just one variable but a second one steering a distribution as well. There is a confidence interval around each y prediction and this is known as a **prediction interval**.

Let's bring back some context with our veterinary example, estimating the age of a dog and number of vet visits. I want to know the prediction interval for number of vet visits with 95% confidence for a dog that is 8.5 years old. What this prediction interval looks like is shown in [Figure 5-16](#). We are 95% confident that an 8.5 year old dog will have had between 16.462 and 25.966 veterinary visits.

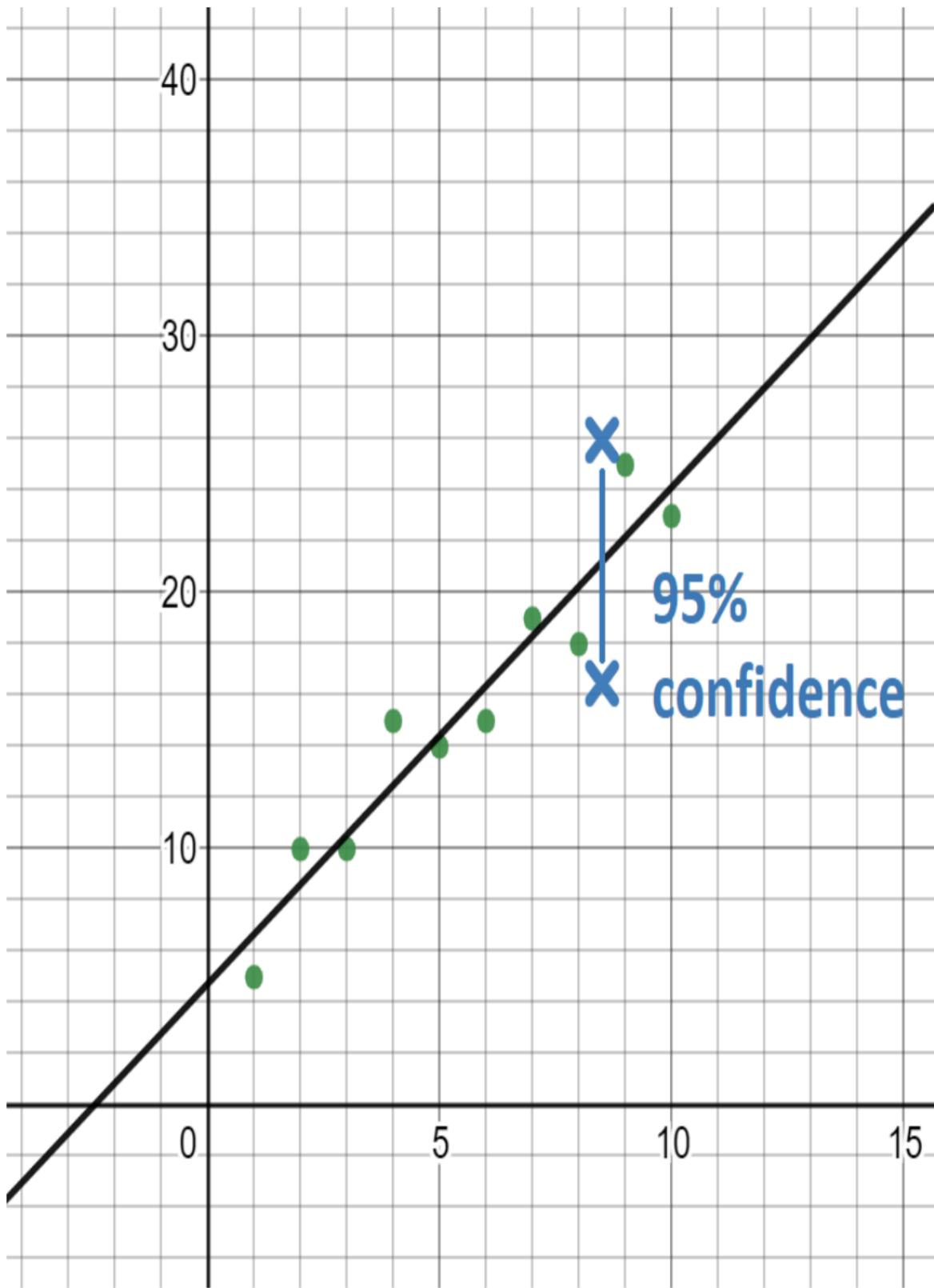


Figure 5-16. A prediction interval for a dog that is 8.5 years old with 95% confidence

How do we calculate this? We need to get the margin of error, and plus/minus it around the predicted y-value. It's a beastly equation that involves a critical value from the t-distribution as well as the standard error of the estimate. Let's take a look:

$$E = t_{.025} * S_e * \sqrt{1 + \frac{1}{n} + \frac{n(x_0 + \bar{x})^2}{n(\sum x^2) - (\sum x)^2}}$$

The x-value we are interested in is specified as x_0 , which in this case is 8.5. Here is how we solve this in Python shown in [Example 5-19](#).

Example 5-19. Calculating a prediction interval of vet visits for a dog that's 8.5 years old

```
import pandas as pd
from scipy.stats import t
from math import sqrt

# Load the data
points = list(pd.read_csv('https://bit.ly/2KF29Bd', delimiter=',').itertuples())

n = len(points)

# Linear Regression Line
m = 1.939
b = 4.733

# Calculate Prediction Interval for x = 8.5
x_0 = 8.5
x_mean = sum(p.x for p in points) / len(points)

t_value = t(n - 2).ppf(.975)

standard_error = sqrt(sum((p.y - (m * p.x + b)) ** 2 for p in points) / (n - 2))

margin_of_error = t_value * standard_error * \
    sqrt(1 + (1 / n) + (n * (x_0 - x_mean) ** 2) / \
        (n * sum(p.x ** 2 for p in points) - sum(p.x for p in points) ** 2))

predicted_y = m*x_0 + b

# Calculate prediction interval
```

```
print(predicted_y - margin_of_error, predicted_y + margin_of_error)
# 16.462516875955465 25.966483124044537
```

Oy vey! That's a lot of calculation, and unfortunately SciPy and other mainstream data science libraries do not do this. But if you are inclined towards statistical analysis, this is very useful information. We not only create a prediction based on a linear regression (e.g. a dog that's 8.5 years old will have 21.2145 vet visits), but we are actually able to say something much less absolute: there's a 95% likelihood an 8.5 year old dog will visit the vet between 16.46 and 25.96 times. Brilliant, right? And it's a much safer claim because it captures a range rather than a single value, and thus accounts for uncertainty.

CONFIDENCE INTERVALS FOR THE PARAMETERS

When you think about it, the linear regression line itself is a sample statistic, and there is a linear regression line for the whole population we are trying to infer. This means parameters like m and b have their own distributions, and we can model confidence intervals around m and b individually to reflect the population's slope and y-intercept. This is beyond the scope of this book but it's worth noting this type of analysis can be done.

You can find resources to do these calculations from scratch but it might just be easier to use Excel's regression tools or libraries for Python. We will talk about specialized libraries for Python at the end of this chapter.

Train/Test Splits

This analysis I just did with the correlation coefficient, statistical significance, and coefficient of determination unfortunately is not done often by machine learning practitioners. Sometimes they are dealing with so much data they do not have the time or technical ability to. For example a 128x128 pixel image is at least 16,384 variables. Do you have time to do statistical

analysis on each of those pixel variables? Probably not! Unfortunately this leads many data scientists to not learn these statistical metrics at all.

In an [obscure online forum](#), I once read a post saying statistical regression is a scalpel, while machine learning is a chainsaw. When operating with massive amounts of data and variables, you cannot sift through all of that with a scalpel. You have to resort to a chainsaw and while you lose explainability and precision, you at least can scale to make broader predictions on more data. That being said, statistical concerns like sampling bias and overfitting do not go away. But there are a few practices that can be employed for quick-and-dirty validation.

WHY ARE THERE NO CONFIDENCE INTERVALS AND P-VALUES IN SCIKIT-LEARN?

Scikit-Learn does not support confidence intervals and P-values, as these two techniques are open problems for higher-dimensional data. This only emphasizes the gap between statisticians and machine learning practitioners. As one of the maintainers of Scikit-Learn, Gael Varoquaux, said “... in general computing correct p-values requires assumptions on the data that are not met by the data used in machine learning (no multicollinearity, enough data compared to the dimensionality) ... P-values are something that are expected to be well checked (they are a guard in medical research). Implementing them is asking for trouble... We can give p-values only in very narrow settings [with few variables].”

If you want to go down the rabbit hole, there are some interesting discussions on Github:

<https://github.com/scikit-learn/scikit-learn/issues/6773>

<https://github.com/scikit-learn/scikit-learn/issues/16802>

As mentioned before, `statsmodel` is a library that provides helpful tools for statistical analysis. Just know it will likely not scale to larger dimensional models because of the aforementioned reasons.

<https://www.statsmodels.org/stable/regression.html>

A basic technique machine learning practitioners will use to mitigate overfitting is a practice called the **train/test split**, where typically 1/3 of the data is set aside for testing and the other 2/3 is used for training (other ratios can be used as well). The **training dataset** is used to fit the linear regression, while the **testing dataset** is used to measure the linear

regression's performance on data it has not seen before. This technique is generally used for all supervised machine learning including logistic regression and neural networks. [Figure 5-17](#) shows a visualization of how we break up our data into 2/3 for training and 1/3 for testing.

DATA

TRAINING
TESTING

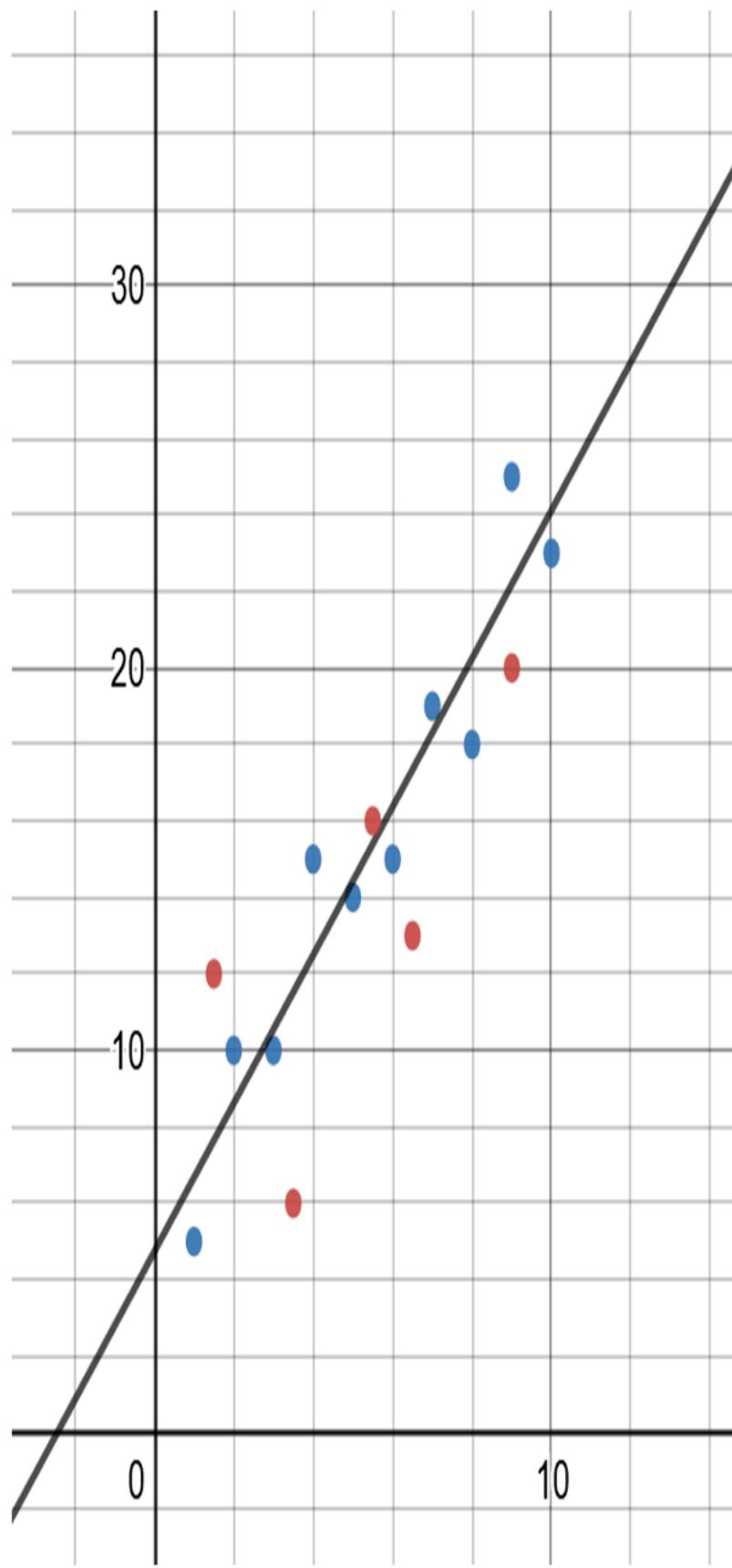


Figure 5-17. Splitting into training/testing data. The line is fitted to the training data (blue) using least squares, while the test data (red) is analyzed afterwards to see how off the predictions are on data that was not seen before

Example 5-20 shows how to perform a train/test split using Scikit-Learn, where 1/3 of the data is set aside for testing, and the other 2/3 is used for training.

TRAINING IS FITTING A REGRESSION

Remember that “fitting” a regression is synonymous with “training”. The latter word is used by machine learning practitioners.

Example 5-20. Doing a train/test split on linear regression

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# Load the data
df = pd.read_csv('https://bit.ly/3cIH97A', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)
Y = df.values[:, -1]

# Separate training and testing data
# This leaves a third of the data out for testing
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=1/3)

model = LinearRegression()
model.fit(X_train, Y_train)
result = model.score(X_test, Y_test)
print("R^2: %.3f" % result)
```

Notice that the `train_test_split()` will take our dataset (X and Y columns), shuffle it, and then return our training and testing datasets based on our testing dataset size. We use the `LinearRegression`'s `fit()` function to fit on the training data `X_train` and `Y_train`. Then we use the `score()`

function on testing datasets `X_test` and `Y_test` to evalute the r^2 , giving us a sense how the regression performs on data it has not seen before. The higher the r^2 is for our test dataset, the better. Having that number higher indicates the regression performs well on data it has not seen before.

USING R-SQUARE FOR TESTING

Note the r^2 is calculated a little bit different here as we have a predefined linear regression from training. We are comparing the testing data to a regression line built from training data. The objective is still the same: being closer to 1.0 is desirable to show the regression correlation is strong even with the training data, while being closer to 0.0 indicates the test dataset performs poorly. Here is how it is calculated, where y_i is each actual y-value, \hat{y}_i is each predicted y-value, and \bar{y} is the average y-value for all data points.

$$r^2 = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2}$$

Figure 5-18 shows different r^2 values from different linear regressions.



$$R^2 = 1.0$$

(PERFECT!)



$$R^2 = 0.917$$

(GOOD!)



$$R^2 = 0.071$$

(POOR!)

Figure 5-18. An R-square applied to different testing datasets with a trained linear regression

We can also alternate the testing dataset across each 1/3 fold. This is known as **cross-validation** and is often considered the gold standard of validation techniques. [Figure 5-19](#) shows how each 1/3 of the data takes a turn being the testing dataset.

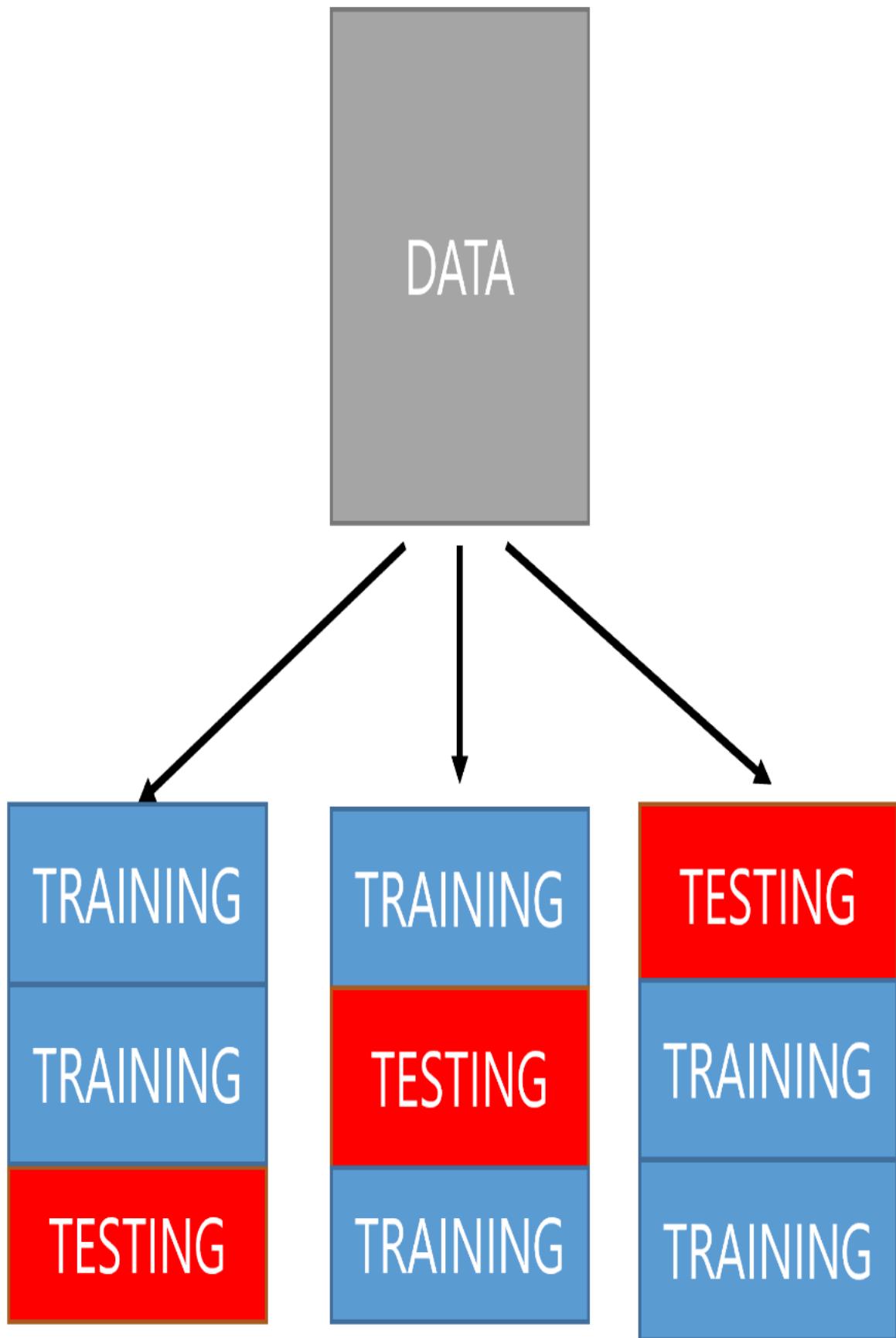


Figure 5-19. A visualization of cross-validation with three folds

The code in [Example 5-21](#) shows a cross-validation performed across three folds, and then the scoring metric (in this case the mean sum of squares (MSE)) is averaged alongside its standard deviation to show how consistently each test performed.

Example 5-21. Using 3-fold cross validation for a linear regression

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import KFold, cross_val_score

df = pd.read_csv('https://bit.ly/3cIH97A', delimiter=',')

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)
Y = df.values[:, -1]

# Perform a simple linear regression
kfold = KFold(n_splits=3, random_state=7, shuffle=True)
model = LinearRegression()
results = cross_val_score(model, X, Y, cv=kfold)
print(results)
print("MSE: mean=%.3f (stdev=%.3f)" % (results.mean(), results.std()))
```

DO TRAINING/TEST SPLITS HAVE TO BE THIRDS?

We do not have to fold our data by thirds. You can use **k-fold validation** to split on any proportion. Typically 1/3, 1/5, or 1/10 are commonly used for the proportion of test data, but 1/3 is the most common.

Generally, the k you choose has to result in a test dataset that has a large enough sample for the problem. You also need enough alternating/shuffled test datasets to provide a fair estimate of performance on data that was not seen before. Modestly sized datasets can use k values of 3, 5, or 10. **Leave-one-out-cross-validation (LOOCV)** will alternate each individual data record as the only sample in the test dataset, and this can be helpful when the whole dataset is small.

When you get concerned about variance in your model, one thing you can do is rather than doing a simple train/test split or cross-validation, you can use **random fold validation** to repeatedly shuffle and train/test split your data an unlimited number of times and aggregate the testing results. In [Example 5-22](#) there are 10 iterations of randomly sampling a 1/3 of the data for testing and the other 2/3 for training. Those 10 testing results are then averaged alongside its standard deviation to see how consistently the test datasets perform.

What's the catch? It's computationally very expensive as we are training the regression many times.

Example 5-22. Using a random-fold validation for a linear regression

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score, ShuffleSplit

df = pd.read_csv('https://bit.ly/38XwbeB', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column) |
```

```

Y = df.values[:, -1]

# Perform a simple linear regression
kfold = ShuffleSplit(n_splits=10, test_size=.33, random_state=7)
model = LinearRegression()
results = cross_val_score(model, X, Y, cv=kfold)

print(results)
print("mean=% .3f (std=% .3f)" % (results.mean(), results.std()))

```

So when you are crunched for time or your data is too voluminous to statistically analyze, a training/testing split is going to provide a way to measure how well your linear regression will perform on data it has not seen before.

TRAIN/TEST SPLITS ARE NOT GUARANTEES

It is important to note that just because you apply machine learning best practices of splitting your training and testing data, it does not mean your model is going to perform well. You can easily over-tune your model and p-hack your way into a good test result, only to find it does not work well out in the real-world. This is why holding out another dataset called the **validation set** is sometimes necessary, especially if you are comparing different models or configurations. That way your tweaks on the training data to get better performance on the testing data does not leak info into the training. You can use the validation dataset as one last stopgap to see if p-hacking caused you to overfit to your testing dataset.

Even then, your whole dataset (including training, testing, and validation) could have been biased to begin with, and no split is going to mitigate that. Andrew Ng discussed this as a large problem with machine learning during his Q&A with DeepLearning.AI and Stanford HAI. He walked through an example showing why machine learning has not replaced radiologists (<https://spectrum.ieee.org/andrew-ng-x-rays-the-ai-hype>).

Multiple Linear Regression

We put almost exclusive focus on doing linear regression on one input variable and one output variable in this chapter. However the concepts we learned here should largely apply to multivariable linear regression. Metrics like r^2 , standard error, and confidence intervals can be used but it gets harder with the more data/variables you have. [Example 5-23](#) is an example

of a linear regression with two input variables and one output variable using Scikit-Learn.

Example 5-23. A linear regression with two input variables

```
import pandas as pd
from sklearn.linear_model import LinearRegression

# Load the data
df = pd.read_csv('https://bit.ly/2X1HWH7', delimiter=',')

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)
Y = df.values[:, -1]

# Training
fit = LinearRegression().fit(X, Y)

# Print coefficients
print("Coefficients = {}".format(fit.coef_))
print("Intercept = {}".format(fit.intercept_))
print("z = {} + {}x + {}y".format(fit.intercept_, fit.coef_[0], fit.coef_[1]))
```

There is a degree of precariousness when a model becomes so inundated with variables it starts to lose explainability, and this is when machine learning practices start to come in and treat the model as a black box. I hope that you are convinced statistical concerns do not go away, and data becomes increasingly sparse the more variables you add. But if you step back and analyze the relationships between each pair of variables using a correlation matrix, and seek understanding on how each pair of variables interact, it will only help your efforts creating a productive machine learning model.

Conclusions

We covered a lot in this chapter. We attempted to go beyond a cursory understanding of linear regression and making train/test splits our only validation. I wanted to show you both the scalpel (statistics) and the chainsaw (machine learning) so you can judge which is best for a given problem you encounter. There are a lot of metrics and analysis methods available in linear regression alone, and we covered a number of them to

understand whether a linear regression is reliable for predictions. You may find yourself in positions to do regressions fast and dirty (via machine learning) or meticulously analyze and comb your data using statistical tools. Which approach you use is situational, and if you want to learn more about statistical tools available for Python check out the `statsmodel` library.

<https://www.statsmodels.org/stable/regression.html>

Throughout the rest of this book I scale back on doing detailed statistical analysis of the models we create, and take a more machine-learning oriented approach using train/test splits as well as confusion matrices. In Chapter 6 covering logistic regression, we will still apply the R^2 and statistical significance. But I hope that this chapter convinced you there are ways to analyze data meaningfully, and the investment can make the difference between a successful project and a rushed one.

Exercises

A dataset of two variables, x and y , is provided here:

<https://bit.ly/3C8JzrM>.

1. Perform a simple linear regression to find the m and b values that minimizes the loss (sum of squares).
2. Calculate the correlation coefficient and statistical significance of this data. Is the correlation useful?
3. If I predict where $x = 50$, what is the 95% prediction interval for the predicted value of y ?
4. Start your regression over and do a train/test split. Feel free to experiment with cross-validation and random-fold validation. Does the linear regression perform well and consistently on the testing data? Why or why not?

About the Author

Thomas Nield is the founder of Nield Consulting Group as well as an instructor at O'Reilly Media and University of Southern California. He enjoys making technical content relatable and relevant to those unfamiliar or intimidated by it. Thomas regularly teaches classes on data analysis, machine learning, mathematical optimization, and practical artificial intelligence. He's authored two books, including *Getting Started with SQL* (O'Reilly) and *Learning RxJava* (Packt).