

*Michael Clark & Seth Berry*

---

# **Models Demystified**

*A Practical Guide from t-tests to Deep Learning*

---

---

## *Contents*

---

<b>Preface</b>	<b>ix</b>
<b>Preface</b>	<b>ix</b>
Acknowledgments . . . . .	ix
<b>1 Introduction</b>	<b>1</b>
1.1 What Is This Book? . . . . .	1
1.2 Who Should Use This Book? . . . . .	2
1.3 What Can You Expect? . . . . .	3
1.4 Which Language? . . . . .	4
1.5 Choose Your Own Adventure . . . . .	5
1.5.1 Overview . . . . .	5
1.5.2 Key Ideas . . . . .	5
1.5.3 Demonstration . . . . .	6
1.5.4 Visualization . . . . .	10
1.5.5 Commentary . . . . .	12
1.6 Moving Towards An Excellent Adventure . . . . .	12
<b>2 The Foundation</b>	<b>15</b>
2.1 Introducing the Greatest Of All Time . . . . .	15
2.2 Key Ideas . . . . .	15
2.2.1 Why this matters . . . . .	16
2.2.2 Good to know . . . . .	16
2.3 What is a Model? . . . . .	16
2.4 What goes into a model? . . . . .	17
2.4.1 Features and Targets . . . . .	17
2.4.2 Expressing Relationships . . . . .	18
2.5 <i>THE</i> Linear Model . . . . .	19
2.5.1 The Linear Model as a Graph . . . . .	22
2.6 What do we do with a model? . . . . .	23
2.6.1 Prediction . . . . .	24
2.6.2 What kinds of predictions can we get? . . . . .	25
2.6.3 Prediction Error . . . . .	27
2.7 How do we interpret the model? . . . . .	30
2.7.1 Feature Level . . . . .	30
2.7.2 Is it a Good Model? . . . . .	33

2.7.3	Prediction vs. Explanation . . . . .	35
2.8	Adding Complexity . . . . .	36
2.8.1	Multiple Features . . . . .	37
2.8.2	More Interpretation Tools . . . . .	42
2.8.3	Feature Importance . . . . .	47
2.8.4	Model Level Interpretation . . . . .	50
2.8.5	Other Complexity . . . . .	50
2.9	Assumptions and More . . . . .	55
2.9.1	More Complex Models . . . . .	57
2.10	Classification . . . . .	57
2.11	More linear models . . . . .	58
2.12	Wrapping Up . . . . .	59
2.12.1	Choose your own adventure . . . . .	60
2.13	Exercise . . . . .	60
2.14	refs . . . . .	61
<b>3</b>	<b>Knowing Your Model</b>	<b>63</b>
3.1	Key Ideas . . . . .	63
3.1.1	Why this matters . . . . .	63
3.1.2	Good to know . . . . .	64
3.2	Model Metrics . . . . .	64
3.2.1	Regression Metrics . . . . .	64
3.2.2	Classification Metrics . . . . .	70
3.3	Model Visualizations . . . . .	78
3.3.1	Regression . . . . .	79
3.3.2	Classification . . . . .	85
3.4	Wrapping Up . . . . .	88
3.5	Additional Resources . . . . .	88
<b>4</b>	<b>How Did We Get Here?</b>	<b>89</b>
4.1	Key Ideas . . . . .	90
4.1.1	Why this matters . . . . .	90
4.1.2	Good to know . . . . .	90
4.2	Data Setup . . . . .	91
4.3	Starting Out by Guessing . . . . .	93
4.4	Prediction Error . . . . .	93
4.5	Ordinary Least Squares . . . . .	95
4.6	Optimization . . . . .	99
4.7	Maximum Likelihood . . . . .	102
4.8	Estimation: Quick Review . . . . .	111
4.9	Penalized Objectives . . . . .	112
4.10	Classification . . . . .	115
4.10.1	Misclassification . . . . .	115
4.10.2	Log loss . . . . .	116
4.11	Optimization Algorithms . . . . .	119

<i>Contents</i>	iii
4.11.1 Gradient Descent . . . . .	119
4.11.2 Stochastic Gradient Descent . . . . .	123
4.11.3 Other Optimization Algorithms . . . . .	128
4.12 Other Estimation Approaches . . . . .	129
4.12.1 Bootstrap . . . . .	130
4.12.2 Bayesian Estimation . . . . .	133
4.13 Wrapping Up . . . . .	138
4.14 Where to Go From Here . . . . .	138
4.15 Exercise . . . . .	139
<b>5 Generalized Linear Models</b>	<b>141</b>
5.1 Key Ideas . . . . .	141
5.1.1 Why this matters . . . . .	141
5.1.2 Good to know . . . . .	142
5.2 Distributions & Link Functions . . . . .	142
5.3 Logistic Regression . . . . .	142
5.3.1 Why Should You Care . . . . .	142
5.3.2 The Binomial Distribution . . . . .	143
5.3.3 Probability, Odds, and Log Odds . . . . .	145
5.3.4 Data Import and Preparation . . . . .	145
5.3.5 Standard Functions . . . . .	147
5.3.6 Interpretation and Visualization . . . . .	148
5.3.7 Objective Function . . . . .	151
5.3.8 Model Fitting . . . . .	151
5.4 Poisson Regression . . . . .	153
5.4.1 Why Should You Care . . . . .	153
5.4.2 The Poisson Distribution . . . . .	153
5.4.3 The (Sometimes) Thin Line . . . . .	155
5.4.4 Standard Functions . . . . .	156
5.4.5 Model Specification . . . . .	159
5.4.6 Model Fitting . . . . .	160
5.5 Wrapping Up . . . . .	161
5.6 Additional Resources . . . . .	161
<b>6 Extending the Linear Model</b>	<b>163</b>
6.1 Key Ideas . . . . .	163
6.1.1 Why this matters? . . . . .	163
6.1.2 Good to know . . . . .	164
6.2 Quantile Regression . . . . .	164
6.2.1 Why Should You Care? . . . . .	164
6.2.2 When The Mean Breaks Down . . . . .	164
6.2.3 Data Import and Preparation . . . . .	167
6.2.4 Standard Functions . . . . .	168
6.2.5 Quantile Loss Function . . . . .	170
6.2.6 Model Fitting . . . . .	172

6.3	Additive Models . . . . .	172
6.3.1	Why Should You Care? . . . . .	173
6.3.2	When Straight Lines Aren't Enough . . . . .	173
6.3.3	Standard Functions . . . . .	175
6.3.4	Splines . . . . .	177
6.3.5	Model Matrix Function . . . . .	178
6.3.6	Model Matrix . . . . .	179
6.3.7	Model Fitting . . . . .	180
6.3.8	Prediction . . . . .	181
6.3.9	Penalized Cubic Spline . . . . .	182
6.3.10	Penalized Model Fitting Function . . . . .	183
6.3.11	Penalized Model Fitting . . . . .	184
6.4	Mixed Models . . . . .	186
6.4.1	Why Should You Care? . . . . .	186
6.4.2	Knowing Your Data . . . . .	186
6.4.3	Standard Functions . . . . .	190
6.4.4	Model Matrix . . . . .	195
6.4.5	Likelihood Function . . . . .	196
6.4.6	Model Fitting . . . . .	197
6.5	Performance Comparisons . . . . .	198
6.6	Wrapping Up . . . . .	198
6.7	Next Steps . . . . .	199

## I Linear Models 13

7	Core Concepts <span style="float: right;">203</span>
7.1	Key ideas . . . . . <span style="float: right;">204</span>
7.1.1	Why this matters . . . . . <span style="float: right;">204</span>
7.1.2	Good to know . . . . . <span style="float: right;">205</span>
7.2	Objective Functions . . . . . <span style="float: right;">205</span>
7.3	Performance Metrics . . . . . <span style="float: right;">206</span>
7.4	Generalization . . . . . <span style="float: right;">210</span>
7.4.1	Using Metrics for Model Evaluation and Selection . . . . . <span style="float: right;">212</span>
7.4.2	Understanding Test Error and Generalization . . . . . <span style="float: right;">213</span>
7.5	Regularization . . . . . <span style="float: right;">215</span>
7.6	Cross-validation . . . . . <span style="float: right;">219</span>
7.6.1	Methods of Cross-validation . . . . . <span style="float: right;">222</span>
7.7	Tuning . . . . . <span style="float: right;">223</span>
7.7.1	A Tuning Example . . . . . <span style="float: right;">224</span>
7.8	Pipelines . . . . . <span style="float: right;">228</span>
7.9	Commentary . . . . . <span style="float: right;">230</span>
7.10	Using R and Python in ML . . . . . <span style="float: right;">231</span>
7.10.1	Python . . . . . <span style="float: right;">231</span>
7.10.2	R . . . . . <span style="float: right;">232</span>
7.11	Where to go from here . . . . . <span style="float: right;">233</span>

7.11.1 refs . . . . .	233
<b>8 Common Models</b>	<b>235</b>
8.1 Key Ideas . . . . .	235
8.1.1 Why this matters . . . . .	235
8.1.2 Good to know . . . . .	236
8.2 General Approach . . . . .	236
8.3 Data setup . . . . .	237
8.4 Do Better than the Baseline . . . . .	238
8.4.1 Why do we do this? . . . . .	238
8.5 Penalized Linear Models . . . . .	239
8.5.1 Elastic Net . . . . .	240
8.5.2 Strengths & Weaknesses . . . . .	241
8.5.3 Additional Thoughts . . . . .	242
8.6 Tree-based methods . . . . .	242
8.7 Deep Learning and Neural Networks . . . . .	247
8.7.1 What is a neural network? . . . . .	247
8.7.2 Trying it out . . . . .	248
8.8 A Tuned Example . . . . .	252
8.9 Comparing models . . . . .	254
8.10 Interpretation . . . . .	256
8.10.1 Feature Importance . . . . .	256
8.11 Other ML Models for Tabular Data . . . . .	258
8.12 Wrapping Up . . . . .	259
8.13 Exercise . . . . .	259
8.14 Where to go from here . . . . .	260
8.14.1 refs . . . . .	260
<b>9 More ML</b>	<b>261</b>
9.1 Key Ideas . . . . .	261
9.1.1 Why this matters . . . . .	261
9.1.2 Good to know . . . . .	261
9.2 Unsupervised Learning . . . . .	262
9.2.1 Connections . . . . .	263
9.2.2 Other classical unsupervised learning techniques . . . . .	267
9.3 Reinforcement Learning . . . . .	268
9.4 Non-Tabular Data Applications . . . . .	269
9.4.1 Spatial . . . . .	270
9.4.2 Audio . . . . .	270
9.4.3 Image Processing . . . . .	270
9.4.4 Natural Language Processing . . . . .	271
9.4.5 Pre-trained Models & Transfer Learning . . . . .	272
9.4.6 Combining Models . . . . .	273
9.5 Artificial Intelligence . . . . .	273
9.6 Where to go from here . . . . .	275

<b>II Machine Learning</b>	<b>201</b>
<b>10 Data Issues in Modeling</b>	<b>279</b>
10.1 Key Ideas . . . . .	279
10.2 Standard Feature & Target Transformations . . . . .	279
10.2.1 Numeric variables . . . . .	280
10.2.2 Categorical variables . . . . .	282
10.2.3 Ordinal Variables . . . . .	283
10.3 Missing Data . . . . .	285
10.4 Class Imbalance . . . . .	287
10.4.1 Calibration issues in classification . . . . .	289
10.5 Censoring and Truncation . . . . .	290
10.6 Time Series . . . . .	293
10.6.1 Time-based Features . . . . .	294
10.7 Spatial Data . . . . .	295
10.8 Latent Variables . . . . .	296
10.9 Commentary . . . . .	297
10.10refs . . . . .	297
<b>11 Causal Modeling</b>	<b>299</b>
11.1 Key ideas . . . . .	299
11.2 Why it matters . . . . .	300
11.2.1 Good to know . . . . .	300
11.3 Classic Experimental Design . . . . .	300
11.4 Natural Experiments . . . . .	301
11.5 Causal Inference . . . . .	302
11.6 Models for Causal Inference . . . . .	302
11.6.1 Linear Regression . . . . .	302
11.6.2 Structural Equation Models . . . . .	303
11.6.3 Counterfactual Thinking . . . . .	303
11.6.4 Uplift Modeling . . . . .	304
11.6.5 Meta-Learning . . . . .	305
11.6.6 Others approaches . . . . .	306
11.7 Commentary . . . . .	306
11.8 Where to go from here . . . . .	307
<b>12 Misc Models</b>	<b>309</b>
<b>III Other Considerations</b>	<b>277</b>
<b>A References</b>	<b>311</b>
<b>B Dataset Descriptions</b>	<b>315</b>
B.1 Heart Failure . . . . .	315
B.2 Heart Disease UCI . . . . .	316

<i>Contents</i>	vii
<b>C Matrix Operations</b>	<b>317</b>
C.1 Addition . . . . .	318
C.2 Subtraction . . . . .	319
C.3 Transpose . . . . .	321
C.4 Multiplication . . . . .	321
C.5 Inversion . . . . .	324
C.5.1 Matrix Determinant . . . . .	324
<b>D Other to come</b>	<b>327</b>
D.1 Simulation . . . . .	327
D.2 Bayesian Demonstration . . . . .	327
D.3 Linear Programming . . . . .	330



---

## **Preface**

---

TODO: Quarto bug prints this twice in ToC; tried various header tricks to no avail. Also, not unnumbered by default for krantz.cls.

Hi there, this is our book! It's about models great and small, and we hope you'll find some useful things here. Our main goal is to provide a resource for people who are learning about models, and we hope that it will be useful for people who are just starting out, as well as for people who are already familiar with models but want to learn from a different perspective. Whether you're a machine learning master that would like a little bit more statistical nuance, or an academic trying to dive into the world of machine learning, we hope you'll find it helpful.

You'll definitely want to have some familiarity with R or Python, and some basic knowledge of statistics will be helpful. We'll try to explain things as we go, but we won't be able to cover everything. If you're looking for a good introduction to R, we recommend *R for Data Science*<sup>1</sup> or the *Python Data Science Handbook*<sup>2</sup> for Python. Beyond that, we'll try to provide the context you need. We're not here to make you an expert, just to help you get acquainted with the world of models.

---

## **Acknowledgments**

A lot of people helped when writing the book...

---

<sup>1</sup><https://r4ds.had.co.nz/>

<sup>2</sup><https://jakevdp.github.io/PythonDataScienceHandbook/>



# 1

---

## *Introduction*

---

Regardless of background, and whether we're conscious of it or not, we are constantly inundated with data. It's inescapable, from our first attempts to understand the world around us, to our most recent efforts to explain why we still don't get it. Even now, our most complicated and successful models are almost uninterpretable even to those that created them. But that doesn't mean that even in those cases we can't understand the essence of how they work. And if you're reading this, you are probably the type of person that wants to keep trying! So for seasoned professionals or perhaps just the data curious, we want to help you learn more about how to use data to answer the questions you have.

---

### **1.1 What Is This Book?**

This book is a practical resource, something we hope you can refer to for a quick overview of a specific modeling technique, a reminder of something you've seen before, or perhaps a sneak peak into some modeling details. The text is focused on a few statistical and machine learning approaches that are widely employed, and specifically those which form the basis for most other models in use. Believe it or not, whether a lowly *t*-test or a complex neural network, there is a tie that binds. We hope to help you understand some of the core modeling principles, and how the simpler models can be extended and applied to a wide variety of data scenarios.

Our approach here is first and foremost a practical one, as models themselves are just tools to help us reach a goal. If a model doesn't work in the world, it's not very useful. But modeling is often a delicate balance of interpretation and prediction, and each data situation is unique in some way, requiring a bespoke approach. What works well in one setting may be poor in another, and what may be the state of the art may only be marginally better than a notably simpler approach that is far more interpretable. In addition, complexities arise even in an otherwise deceptively simple application. However, if you have the core understanding of the techniques that lie at the heart of many models, you'll automatically have many more tools at your disposal to tackle

the problems you face, and be more comfortable with choosing the best for your needs.

---

## 1.2 Who Should Use This Book?

This book is intended for every type of *data dabbler*, no matter what part of the data world you call home. If you consider yourself a data scientist, a business analyst, or a statistical hobbyist, you already know that the best part of a good dive into data is the modeling. But whatever your data persuasion, models give us the possibility to answer questions, make predictions, and understand what we're interested in a little bit better. And no matter who you are, it isn't always easy to understand *how the models work*. Even when you do get a good grasp of a modeling approach, it can still get complicated, and there are a lot of details to keep track of. In other cases, maybe you just have other things going on in your life and have forgotten a few things. We find that it's always good to remind yourself of the basics! So if you're just interested in data and hoping to understand it a little better, then it's likely you'll find something useful in this book!

Your humble authors have struggled mightily themselves throughout the course of their data history, and still do! We were initially taught by those that weren't exactly experts, and often found it difficult to get a good grasp of statistical modeling and machine learning. We've had to learn how to use the tools, how to interpret the results, and possibly the most difficult, how to explain what we're doing to others! We've forgotten a lot, confused ourselves, and made some happy accidents in the process. That's okay! Our goal here is to help you avoid some of those pitfalls, help you understand the basics of how models work, and get a sense of how most modeling endeavors have a lot of things in common.

Whether you enthusiastically pour over formulas and code, or prefer to skip over them, we promise that you don't need to memorize a formula to get a good understanding of modeling and related issues. We are the first to admit that we have long dumped the ability to pull formulas out of our brain folds<sup>1</sup>; however, knowing how those individual pieces work together only helps to deepen your understanding of the model. Typically using code puts the formula into more concrete terms that you can then use in different ways to solidify and expand your knowledge. Sometimes you just need a reminder or want to see what function you'd use. And often, the visualization will reveal even more about what's going on than the formula or the code. In short, there are a lot of tools at your disposal to help learn modeling in a way that works for you. We hope

---

<sup>1</sup>We actually never had this ability.

that anyone that would be interested in the book will find a way to learn things in a manner that suits them best.

There is bit of a caveat. We aren't going to teach you basic statistics or how to program in R or Python. Although there is a good chance you will learn some of it here, you'll have an easier time if you have a very basic understanding of statistics and some familiarity with coding. We will provide some resources for you to learn more about these topics, but we won't be covering them in detail. The ([appendix?](#)) will provide some more information about prerequisites or just stuff that would be good to know. However, we really aren't assuming a lot of conceptual knowledge, and are, if anything, assuming that whatever knowledge you have may be a bit loose or fuzzy. That's okay!

---

### 1.3 What Can You Expect?

For each model that we cover, you can expect the following in terms of content:

- Overview
  - Why it's useful
  - Conceptual example and interpretation
  - Where the model lies in the grand scheme of topics that we will cover
- Key ideas and concepts
  - Brief summary and definition list of concepts or terms
- Demonstration with data, code, results, and visualizations
  - The data will often be simulations as that opens doors for further understanding, or a dataset that hopefully is a little more interesting than mtcars or iris.
  - The demonstrations will provide you the opportunity to get your hands as dirty as you wish. We will present the code in two ways:
    - \* standard functions (e.g., `lm` in R, `ols` in `statsmodels` for Python)
    - \* the steps to recreate the estimation process on your own (or at least something a little more hands-on)
- Commentary, cautions, and where to explore next

We are taking this approach for one reason: so that you can go as deep as you wish. If you are looking for a quick tutorial on helpful models, then you might not find yourself going any deeper than the standard functions (or even getting into the code at all). If you want to really dive into these models, then you might find yourself working through the complete steps. Another approach is to allow yourself some time between the standard functions and complete steps. You could work through the standard functions of every chapter, give it some time to marinate, and then work back through the complete steps. While

we certainly recommend working through the chapters in order, we want to give you the flexibility to choose your own depth within each.

We hope that this book can serve as a “choose your own adventure” statistical reference. Whether you want a surface-level understanding, a deeper dive, or just want to be able to understand what the analysts in your organization are talking about, you will find value in this book. While we assume that you have a basic familiarity with coding, that doesn’t mean that you need to work through every line of code to understand the fundamental principles and use cases of every model.

---

## 1.4 Which Language?

You’ve probably noticed most books, blogs, and courses on data science choose R or Python. While many individuals often take an opinionated approach towards teaching and using one over the other, we eschew dogmatic approaches and language flame wars. R and Python are both great languages (and equally flawed in unique ways), and it is advantageous to be knowledgeable of both, even if you focus on one specifically, as they are the most popular languages for statistical modeling and machine learning. We use both extensively in our own work for teaching, personal use, and production level code, and have found both are up to whatever task you have in mind. Throughout this book, we will be presenting demonstrations in both R and Python, and you can use both or take your pick, but we want to leave that choice up to you. Our goal isn’t to obscure the ideas behind packages and specialty functions or tell you why one language is superior to the other (they aren’t), but to show you the most basic functions behind big model ideas.

While we want to provide you choice, we truly hope that displaying both languages can help people to “convert” from one to the other. We have spent countless hours, slumped over our computers, debugging errors and figuring things out. If we can take away one source of pain for you, that would be great! We’d like to consider this as a resource for the R user, who knows exactly what they want to do in R, but could use a little help translating their R knowledge to Python; we’d also like this book to be a resource for the Python user, who sees the value in R’s statistical modeling abilities.

## 1.5 Choose Your Own Adventure

As an example of how things will go, let's look at the different ways we might express a relationship between two variables. If you want just a little bit of background, you can read through the *Overview* and *Key Ideas*. If you want to see the easy way to use the model, you can work through the *Standard Function* section. If you want to see the method from start to finish, feel free to work through the *Roll Your Own* section. If all of that sounds like a lot of work and you'd rather look at the pretty pictures, that's fine too<sup>2</sup>, just jump to the *Visualization* section.

### 1.5.1 Overview

Correlations provide a means of understanding how, and *if*, two or more things are related. For any two variables or features, we can estimate a single value that signals the strength and direction of the relationship between them. Despite limitations, correlations can be a great way to get a quick understanding of the relationship between two features. One variable could be temperature while another variable is the number of ice cream cones sold, or one is the number of hours spent studying and another variable is the grade on a test, maybe the number of hours spent watching TV versus the number of hours spent exercising. The correlation value will give us information that is similarly interpretable in each case.

### 1.5.2 Key Ideas

Here are key ideas to consider for understanding correlation.

- **Variance:** Two variables must vary if they are to *co-vary*
- **Covariance:** Joint variability of two variables, i.e. how they vary together
- **Interpretation:** Covariance is hard to interpret because the variables are typically on different scales
- **Correlation:** Correlation is a standardized covariance, so it is easier to interpret

Another way to think about correlation is through the lens of *variance covariance*. Covariance is a measure of the joint variability of two variables, i.e. how they *vary* together. If two variables are highly correlated, then they have a high degree of covariance, i.e. they vary together in a meaningful way.

---

<sup>2</sup>We love pretty pictures too! Sometimes it's the best way to bring a lot of ideas together.

### 1.5.3 Demonstration

To demonstrate correlation, we could start by formally defining it. Here is a formula for the Pearson-Product-Moment correlation coefficient. What everyone typically just calls “correlation” or Pearson’s  $r$  is actually the *Pearson Product-Moment Correlation* – that is a lot of words, though, so we will just go with correlation. As we mentioned earlier, we want to give you the choice to dive in as deep as you want.

$$\rho = \frac{\Sigma(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\Sigma(x_i - \bar{x})^2}\sqrt{\Sigma(y_i - \bar{y})^2}} \quad (1.1)$$

In words, starting with the numerator, we are going to subtract the mean of  $x$  from every observation of  $x$ , do the same for  $y$ , multiply them together, and then sum them to produce a single value. If you’ve taken a statistics class before, you might have seen this part of the formula when talking about *covariance* – the joint variability of two variables. In fact, if you just divide the numerator by  $n-1$  you’d get the covariance value. Covariance values, though, are not standardized; therefore, they are a bit tough to interpret. You will get a sign indicating relationship (just like a correlation) and the magnitude tells you the strength of the relationship, but it is hard to get your head around how much stronger a covariance of 4.00 is compared to a covariance of 0.28.

This is where the denominator comes into play. To put it into real terms, we will subtract the mean of  $x$  from every observation of  $x$  (i.e. get the deviations of  $x$ ), square those values, and sum those values to produce a single value. We will do the same for  $y$ , multiply those results, and then take the square root of that product. The denominator might also look familiar to some. The individual pieces-  $\Sigma(x_i - \bar{x})^2$  and  $\Sigma(y_i - \bar{y})^2$  - are computing the *variance* of  $x$  and  $y$ . Taking the square root of variance gives us the *standard deviation*. Basically, we’re getting a combined standard deviation to scale the covariance. Using the standard deviation will mitigate the effects of  $x$  and  $y$  potentially being on different scales; in other words, we want to know how much two variables move together, even if the means and typical movement (deviation) might be very different.

Here is a simplification, depicting correlation as a **standardized covariance**.

$$\rho = \frac{cov(x, y)}{\sigma_x \sigma_y} \quad (1.2)$$

where  $cov(x, y)$  is the covariance between  $x$  and  $y$ , and  $\sigma_x$  and  $\sigma_y$  are the standard deviations of  $x$  and  $y$ , respectively. So putting it all together, we are simply looking at the ratio of raw covariance (i.e., how much two variables “move” together) and standard deviation (i.e., the amount of dispersion within a variable). Just as we mentioned earlier, simpler concepts often tie models

together! What might look like a tricky formula, really just starts with means and deviations, variance and covariance!

As a reminder, this correlation will give us an idea about the **linear relationship** between two continuous variables; we will get a value between -1 and 1, with values closer to 0 indicating that there is no linear relationship between the two variables. As values get closer to 1, we have a **positive correlation** – as values for one variable increase, values for the other variable tend to increase along with it. As the correlation gets closer to -1, we have a **negative correlation** – as values for one variable increase, values for the other variable typically decrease. Correlations can be useful for quickly exploring linear relationships, but let's not get too excited about it – they aren't going to help you answer any big questions! We also don't want to get too carried away with "statistical significance" yet – once samples get large, even small correlations become "significant". Instead, just use correlations to explore the patterns within your data, start getting ideas about interesting relationships that you might find, and leave worries about significance for people with more time on their hands.

### 1.5.3.1 Code

It's often easier to understand a concept by seeing it in action. So let's start by creating some data. We'll create a variable `x`, then make a `y` that will have a linear relationship with it, but also have some random noise. We'll then plot the two variables to see what they look like. To help your understanding, fiddle with the knobs noted.

### 1.5.3.2 R

```
set.seed(seed = 1001)
N = 500
x = rnorm(n = N, mean = 0, sd = 1)

# Fiddle with the .5 and .75.
# The first can also be negative if you like!
y = .5 * x + rnorm(n = N, mean = 0, sd = .75)
```

### 1.5.3.3 Python

```
import numpy as np

np.random.seed(seed = 1001)
N = 500
x = np.random.normal(loc = 0, scale = 1, size = N)

# Fiddle with the .5 and .75.
# The first can also be negative if you like!
```

```
y = .5*x + np.random.normal(loc = 0, scale = .75, size = N)
```

Now check out the plot of those two values:

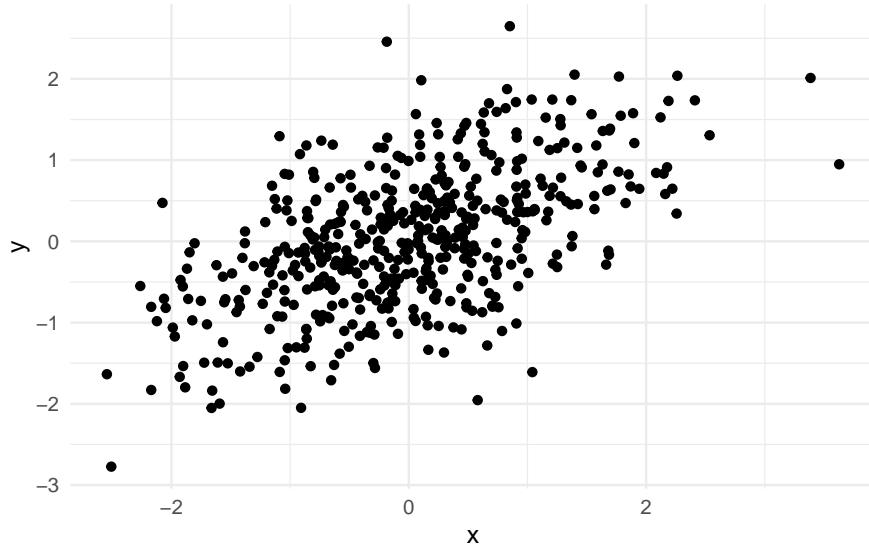


Figure 1.1: Scatterplot of two variables.

Remember that correlation is testing for the presence of a linear relationship, with -1 indicating a perfect negative relationship, 1 indicating a perfect positive relationship, and 0 indicating no relationship. Before we see the actual correlation value for these two variables, take a guess as to what value we are going to get!

Before we create our own function, we can use R's `cor` function or numpy's `corrcoef` function. You should get something around 0.6.

```
# Results for R and Python will be slightly
# different due to different random number generators
cor(x, y)
np.corrcoef(x, y)

[1] 0.557
```

When you guessed the value, were you close? If so, congrats! If not, try fiddling with those knobs noted until things get a little clearer. But now that we already know the answer, let's make sure that we can get the same answer by working through the formula via code. The following takes that initial formula approach and turns it into a function that we can use to compute the correlation between any two variables. We'll then use that function to compute the correlation between our `x` and `y` variables.

### 1.5.3.4 R

```
my_cor = function(x, y) {
  # First, we need to compute the averages for x and y.
  # The rest follows the formula.
  x_bar = mean(x)
  y_bar = mean(y)

  numerator = sum((x - x_bar) * (y - y_bar))

  denominator = sqrt(
    sum((x - x_bar)^2) * sum((y - y_bar)^2)
  )

  numerator / denominator
}

# using the builtin functions
my_cor2 = function(x, y) {
  cov(x, y) / (sd(x) * sd(y))
}

my_cor(x, y)
[1] 0.557
```

### 1.5.3.5 Python

```
def my_cor(x, y):
  # First, we need to compute the averages for x and y.
  # The rest follows the formula.
  x_bar = np.mean(x)
  y_bar = np.mean(y)

  numerator = np.sum((x - x_bar) * (y - y_bar))

  denominator = np.sqrt(
    np.sum((x - x_bar)**2) * np.sum((y - y_bar)**2)
  )

  # We will finish by dividing the numerator by the
  # denominator.
  # This will ensure that we have a value between -1 and 1.
  return(numerator / denominator)

my_cor(x, y)
```

`0.5390318454354402`

It doesn't matter which language we use, the steps are largely the same when we break it down into the individual pieces!

### 1.5.4 Visualization

A long time ago, in a land far away, the authors of this book worked together to help clients traverse the forests of data to reach their modeling goals. While there were many great learning opportunities along the way, working with clients showed us the kinds of help that people really needed in adventuring with data and models. Even so, there were many requests that made us grimace, and one stood atop Mount Ridiculous: to produce a correlation matrix with 115 variables and export that matrix to a spreadsheet. We still don't recommend such shenanigans, but there are ways to try and understand correlation matrices. Since we were in the business of helping people do their work better, one way we often did so was via a *corrplot*.

We'll start with something manageable. We create a data set with six variables of two sets: a, b, c, and x, y, z, and then we can take a quick look at the correlation matrix.

Table 1.1: Correlation matrix

feature	a	b	c	x	y	z
a	1.00	0.46	0.45	-0.16	-0.21	-0.19
b	0.46	1.00	0.49	-0.07	-0.18	-0.14
c	0.45	0.49	1.00	-0.16	-0.23	-0.20
x	-0.16	-0.07	-0.16	1.00	0.49	0.48
y	-0.21	-0.18	-0.23	0.49	1.00	0.52
z	-0.19	-0.14	-0.20	0.48	0.52	1.00

Now we have the *pairwise correlations* between all six of our variables, with 1's on the diagonals (naturally, a variable has a perfect correlation with itself). You can check out the lower diagonal or the upper diagonal, because they contain the exact same information. Quickly, though, find the interesting pattern in that matrix!

Producing the correlations between just 6 variables gives us 15 correlation coefficients to examine! You can see that you'll need to spend more than a few seconds on finding the interesting patterns within the data (or if there are any patterns at all). Our brains are oriented towards vision, so we can use *preattentive processing* elements, like hue, saturation, and size, to make finding interesting patterns easier.

Since we already have a correlation matrix, we can use various means to find

those patterns, which include visualizing the matrix itself, network graphs and others. Here is one way to do it.

**i** Some recommended R packages for visualizing a correlation matrix include `corr`, `ggcorr`, and `corrplot`. For python, one has options for `seaborn`, `pandas`, `biokit`, and others.

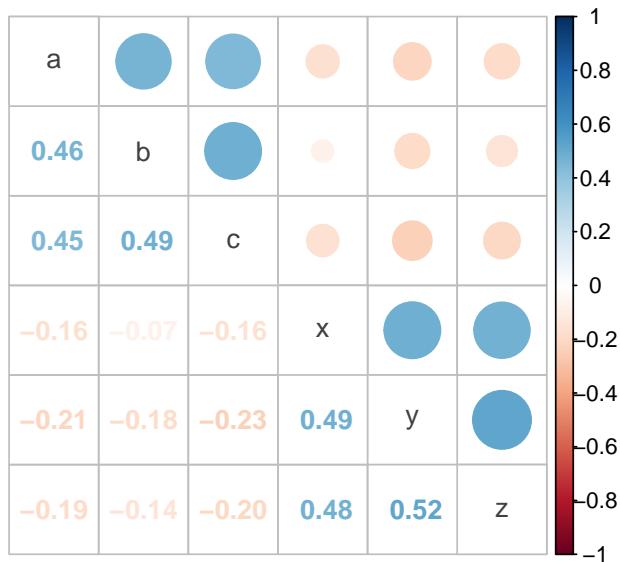


Figure 1.2: Correlation matrix visualization

Let's break down what we're seeing just a little bit. The lower triangle has the correlation values. It adds information, though, by changing the hue by correlation value – red for negative values and blue for positive values – and increasing the saturation as the correlation value becomes stronger. The upper triangle contains the same information, but the size of the circle is tied directly to the strength of the correlation. You'll also notice that the weaker correlations are more hidden in the visualization, allowing us to focus only on those interesting relationships.

What do you think? Was it easier to spot the points of interest? It looks like there is an a-b-c group and x-y-z group that are similarly correlated within their respective groups. We can also see that the a-b-c group is negatively correlated with the x-y-z group. Visualizing the correlation matrix can usually make it easier to find those interesting patterns.

### 1.5.5 Commentary

The correlation is a starting point for understanding linear models, which serve as the foundation for modeling in general. It is very limited by only assessing linear relationships between variables, as well as only pairwise relationships. Other metrics can overcome these, but they have their own limitations. The basic Pearson-Product-Moment correlation coefficient is still the most widely used and a typical starting point in many data adventures.

Things to explore further next:

- Rank correlation (e.g., Spearman’s rho, Kendall’s tau)
- Distance metrics (e.g., euclidean, manhattan, cosine)
- Non-linear relationships and interactions (e.g., distance correlation, polynomial, splines)
- Multivariate relationships (e.g., partial correlations, r-squared)

---

## 1.6 Moving Towards An Excellent Adventure

Remember the point we made about “choosing your own adventure”? Statistical modeling and programming is an adventure, even if you never leave your desk! Every situation calls for choices to be made and every choice you make will lead you down a different path. You will run into errors, dead ends, and you might even find that you’ve spent considerable time to conclude that nothing interesting is happening in your data. This, no doubt, is part of the fun and all of those struggles make success that much sweeter. Like every adventure, things might not be immediately clear and you might find yourself in perilous situations! If you find that something isn’t making sense upon your first read, that is okay. Both authors have spent considerable time mulling over models and foggy ideas during our assorted (mis)adventures; nobody should expect to master complex concepts on a single read through! In any arena where you strive to develop skills, distributed practice and repetition are essential. When concepts get tough, step away from the book, and come back with a fresh mind.

Thanks for coming on this adventure with us and welcome to our *Book of Models*.

This is a test reference (`hastie_elements_2009?`).

---

---

# **Part I**

# **Linear Models**

---

---



# 2

---

## *The Foundation*

---

It is the chief characteristic of data science that it works. Isaac Asimov  
(paraphrased)

---

### 2.1 Introducing the Greatest Of All Time

Now that you're here, it's time to dive in! We'll start things off by covering the building block of all modeling, and a solid understanding here will provide you the basis for just about anything that comes after, no matter how complex it gets. The **linear model** is our starting point. At first glance, it may seem like a very simple model, and it is in some ways, relatively speaking. But it's also quite powerful and flexible, able to take in different types of inputs, handle nonlinear relationships, temporal and spatial relations, clustering, and more. Linear models have a long history, with even the formal and scientific idea behind correlation and linear regression being well over a century old<sup>1</sup>! And in that time, the linear model is far and away the most used model out there. But before we start talking about the *linear* model, we need to talk about what a **model** is in general.

---

### 2.2 Key Ideas

To get us started, we can pose a few concepts key to understanding models. We'll cover each of these as we go along.

- What a model is: The model as an idea
- Features, targets, and input-output mappings: how do we get from input to output?
- Prediction: how do we use a model?

---

<sup>1</sup>Regression in general is typically attributed to Galton<sup>2</sup>, and correlation to Pearson, whose coefficient bearing his name is still the mostly widely used measure of association. Peirce & Bowditch were actually ahead of both (Rovine and Anderson 2004).

- Interpretation: what does a model tell us?
  - Prediction underlies all interpretation
  - We can interpret a model at the feature level and as a whole

As we go along and cover these concepts, be sure that you feel you have the ‘gist’ of what we’re talking about. Almost everything of what comes after linear models builds on these ideas, so it’s important to have a firm grasp before climbing to new heights.

### 2.2.1 Why this matters

The basic linear model and how it comes about underpins so many models, from the simplest t-test to the most complex neural network. It takes a bit to get used to the important aspects of it, but it provides a powerful foundation, and one that you’ll see in many different contexts. It’s also a model that is relatively easy to understand, and one that you can use to understand other models. So it’s a great place to start!

### 2.2.2 Good to know

We’re just starting out here, but we’re kind of assuming you’ve had some exposure to the idea of statistical or other models, even if only from an interpretation standpoint. We assume you have understanding of basic stats like central tendency (e.g., a mean or median) and correlation, stuff like that. And if you intend to get into the code examples, you’ll need a basic familiarity to with Python or R.

---

## 2.3 What is a Model?

At its core, a model is just an **idea**. It’s a way of thinking about the world, about how things work, how things change over time, how things are different from each other, and how they are similar. The underlying thread is that **a model expresses relationships** about things in the world around us. One can also think of a **model as a tool**, one that allows us to take in information, derive some meaning from it, and act on it in some way. Just like other ideas and tools, models have consequences in the real world, and they can be used wisely or foolishly.

On a practical level, a model is expressed through a particular language, math, but don’t let that worry you if you’re not so inclined. As it’s still just an idea at its core, the idea is the most important thing to understand about a model. The **math is just a formal way of expressing the idea** in a manner that can be communicated and understood by others in a standard

way, and math can help make the idea precise. But in everyday terms, we're trying to understand things like how the amount of sleep relates to cognitive functioning, how the weather affects the number of people who visit a park, how much money to spend on advertising to increase sales, how to detect fraud, and so on. Any of these could form the basis of a model, as they stem from scientifically testable ideas, and they all express relationships between things we are interested in, possibly even with an implication of causal relations.

Actually applying models to data can be simple. For example, if you wanted to run a linear model to understand the relationship between sleep and cognitive functioning, you might express it in code as:

### 2.3.0.1 R

```
lm(cognitive_functioning ~ sleep, data = df)
```

### 2.3.0.2 Python

```
from statsmodels.formula.api import ols

model = ols('cognitive_functioning ~ sleep', data = df).fit()
```

The first part with the `~` is the model formula, which is how math comes into play to help us express relationships. Beyond that we just specify where, for example, the scores for cognitive functioning and the amount of sleep is, in this case, in the same data frame called `df`, which may have been imported from a spreadsheet somewhere. Very easy! But that's all it takes to express a straightforward idea. More conceptually, we're saying that cognitive functioning is a linear function of sleep. You can probably already guess why R's function is `lm`, but you'll eventually also learn why `statsmodels` function is `ols`, but for now just know that both are doing the same thing.

## 2.4 What goes into a model?

### 2.4.1 Features and Targets

In the context of a model, how we specify the nature of the relationship depends on the context. In the interest of generality, we'll refer to the **target** as what we want to explain, and **features** as those aspects of the data we will use to explain it. Because people come at data from a variety of contexts, they often use different terminology to mean the same thing. The table below shows some of the common terms used to refer to features and targets.

Table 2.1: Common Terms for Features and Targets

Feature	Target
independent variable	dependent variable
predictor variable	response
explanatory variable	outcome
covariate	label
x	y
input	output
right-hand side	left-hand side

Some of these actually suggest a particular type of relationship (e.g., a causal relationship, an experimental setting), but we'll typically avoid those terms if we can. In the end, we may use any of these words to describe things so that you are comfortable with the terminology, but typically we'll stick with **features** and **targets** for the most part. In our opinion, this terminology has the least hidden assumptions/implications, and just implies ‘features of the data’ and the ‘target’ we’re trying to explain or predict

### 2.4.2 Expressing Relationships

As noted, a model is a way of expressing a relationship between a set of features and a target, and one way of thinking about this is in terms of **inputs** and **outputs**. But how can we go from input to output? Well to begin, we assume that the features and target are **correlated**, that there is some relationship between the feature  $x$  and target  $y$ . If so, then we can ultimately use the features to **predict** the target. In the simplest setting, a correlation implies a relationship where  $x$  and  $y$  typically move up and down together (left plot) or they move in opposite directions where  $x$  goes up and  $y$  goes down (right plot).

In addition, the typical correlation suggests a *linear* relationship, one that is adequately captured by a straight line. There are many types of correlation metrics, but the most common one, the **Pearson correlation**, is explicitly a measure of the linear relationship between two variables. It's expressed as a number between -1 and 1, where 0 means there is no linear relationship. As we move closer to a 1.0 correlation value, we would see a tighter scatterplot like the one on the left, until it became a straight line. The same happens for the negative relationship as we get closer to a value of -1. If we have only one feature and target, the Pearson correlation reflects the exact result of the linear model we'd conduct in a more complicated fashion. But even with multiple features, we often use a version of the Pearson R to help us understand how the features account for the target's variability.

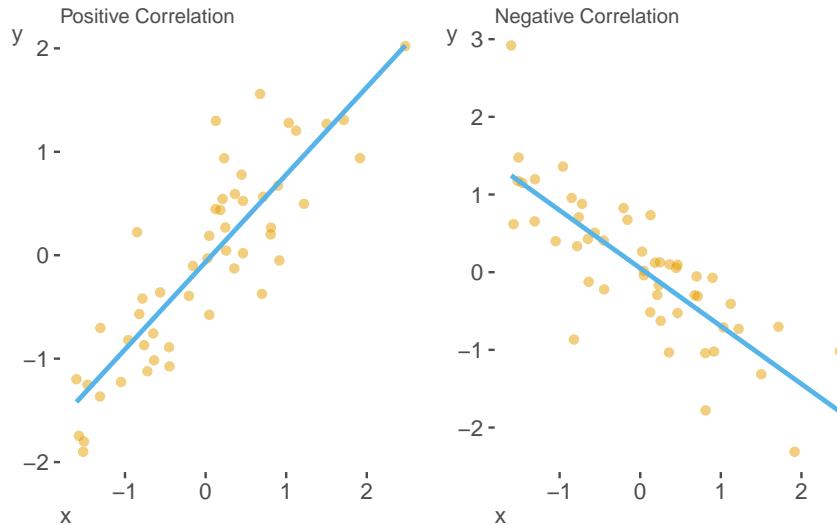


Figure 2.1: Correlation

## 2.5 THE Linear Model

The linear model is perhaps the simplest *functional* model we can use to express a relationship between features and targets. And because of that, it's possibly still the most common model used in practice, and it is the basis for many types of other models. Why don't we run one now?

The following dataset has individual movie reviews containing the movie rating (1-5 stars scale), along with features pertaining to the review (e.g., word count, etc.), those that regard the reviewer (e.g., age) and features about the movie (e.g., genre, release year).

For our first linear model, we'll keep things simple. Let's predict the rating from the length of the review in terms of word count. We'll use the `lm()` function in R and the `ols()` function in Python<sup>3</sup> to fit the model. Both functions take a formula as the first argument, which is a way of expressing the relationship between the features and target. The formula is expressed as  $y \sim x_1 + x_2 + \dots$ , where  $y$  is the target name and  $x$  are the feature names. We also need to specify what the data object is, typically a data frame.

---

<sup>3</sup>We actually are using the `smf.ols` approach because it is modeled on the R approach.

### 2.5.0.1 R

```
df_reviews = read_csv("data/movie_reviews.csv")

model_reviews = lm(rating ~ word_count, data = df_reviews)

summary(model_reviews)

Call:
lm(formula = rating ~ word_count, data = df_reviews)

Residuals:
    Min      1Q  Median      3Q     Max 
-2.0648 -0.3502  0.0206  0.3352  1.8498 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 3.49164   0.04236   82.4   <2e-16 ***  
word_count -0.04268   0.00369  -11.6   <2e-16 ***  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.591 on 998 degrees of freedom
Multiple R-squared:  0.118, Adjusted R-squared:  0.118 
F-statistic: 134 on 1 and 998 DF,  p-value: <2e-16
```

### 2.5.0.2 Python

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

df_reviews = pd.read_csv('data/movie_reviews.csv')

model_reviews = smf.ols('rating ~ word_count', data = df_reviews).fit()

model_reviews.summary(slim = True)
"""


OLS Regression Results
=====
Dep. Variable:          rating    R-squared:       0.118
Model:                 OLS      Adj. R-squared:   0.118
No. Observations:      1000      F-statistic:    134.1
Covariance Type:    nonrobust   Prob (F-statistic): 3.47e-29
=====
            coef    std err        t     P>|t|      [0.025    0.975]
-----

```

```

Intercept      3.4916      0.042     82.431      0.000      3.409      3.575
word_count    -0.0427      0.004    -11.580      0.000     -0.050     -0.035
=====

```

Notes:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.  
"""

For such a simple model, we certainly have a lot to unpack here! Don't worry, you'll eventually come to know what it all means. But it's nice to know how easy it is to get the results! For now we can just say that there's a negative relationship between the word count and the rating, and that the value regarding relationship is statistically significant.

Getting more into the details, we'll start with the fact that the linear model posits a **linear combination** of the features. This is an important concept to understand, but really, a linear combination is just a sum of the features, each of which has been multiplied by some specific value. That value is often called a **coefficient**, or possibly **weight**, depending on the context. The linear model is expressed as (math incoming!):

$$y = b_0 + b_1x_1 + b_2x_2 + \dots + b_nx_n \quad (2.1)$$

- $y$  is the target.
- $x_1, x_2, \dots, x_n$  are the features.
- $b_0$  is the intercept, which is kind of like a baseline value or offset. If we had no features at all it would just be the mean of the target.
- $b_1, b_2, \dots, b_n$  are the coefficients or weights for each feature.

But lets start with something simpler, let's say you want to take a sum of several features. In math you would write it as:

$$x_1 + x_2 + \dots + x_n$$

In the previous equation,  $x$  is the feature and  $n$  is the number identifier for the features, so  $x_1$  is the first feature,  $x_2$  the second, and so on.  $x$  is an arbitrary designation, you could use any letter, symbol you want, or even better, would be the actual feature name. Now look at the linear model.

$$y = x_1 + x_2 + \dots + x_n$$

In this case, the function is *just a sum*, something so simple we do it all the time. In the linear model sense though, we're actually saying a bit more. Another way to understand that equation is that  $y$  is a *function of*  $x$ . We don't show any coefficients here, i.e. the  $bs$  in our initial equation, but technically it's as if each coefficient was a value of 1. In other words, for this simple linear

*model*, we're saying that each feature contributes in an identical fashion to the target.

In practice, features will not contribute in the same ways, because they correlate with the target differently, or are on different scales. So if we want to relate some feature,  $x_1$ , and some other feature,  $x_2$ , to target  $y$ , we probably would not assume that they both contribute in the same way from the beginning. We might give relatively more weight to  $x_1$  than  $x_2$ . In the linear model, we express this by multiplying each feature by a different coefficient. So the linear model is really just a sum of the features multiplied by their coefficients, i.e. a *weighted sum*. In fact, we're saying that each feature contributes to the target in proportion to the coefficient. So if we have a feature  $x_1$  and a coefficient  $b_1$ , then the contribution of  $x_1$  to the target is  $b_1 * x_1$ . If we have a feature  $x_2$  and a coefficient  $b_2$ , then the contribution of  $x_2$  to the target is  $b_2 * x_2$ . And so on. So the linear model is really just a sum of the features multiplied by their respective weights.

For our model, here is the mathematical representation:

$$\text{rating} = b_0 + b_1 \cdot \text{word\_count}$$

And with the actual results of our model:

$$\text{rating} = 3.49 + -0.04 \cdot \text{word\_count}$$

Not too complicated we hope! But let's make sure we see what's going on here just a little bit more.

- Our *idea* is that the length of the review is in some way related to the eventual rating given to the movie.
- Our *target* is rating, and the *feature* is the word count
- We *map the feature to the target* via the linear model, which provides an initial understanding of how the feature is related to the target. In this case, we start with a baseline of 3.49. This value makes sense only in the case of a rating with no review, and is what we would guess if the word count was 0. But we know there are reviews for every observation, so it's not very meaningful as is. We'll talk about ways to get a more meaningful intercept later, but for now, that is our starting point. Moving on, if we add a single word to the review, we expect the rating to go down by -0.04 stars. So if we had a review that was 10 words long, i.e., the mean word count, we would predict a rating of  $3.49 + 10 * -0.04 = 3.1$  stars.

### 2.5.1 The Linear Model as a Graph

We can also express the linear model as a graph, which can be a very useful way to think about models in a visual fashion, and as we see other models, can

help us literally see how different models relate to one another and are actually very similar. In the following, we have three features predicting a single target, so we have three nodes for the features, and a single node for the target. The feature nodes are combined into a linear combination, or, linear predictor, with each ‘edge’ signifying the connection, and labeled with the coefficient or weight. This connection between our linear predictor, and the ultimate target is direct, without any additional change. Later on, we’ll see more about that, but for our standard linear model, we’re all set.

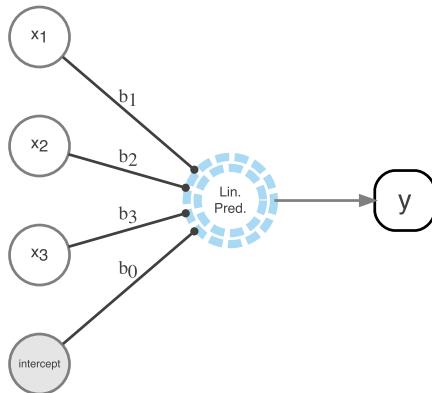


Figure 2.2: A linear regression as a graphical model

So at this point you have the basics of what a linear model is and how it works, and a couple ways to think about it, whether through programming, math, or just visually. But there is a lot more to it than that. Just getting the model is easy enough, but we need to be able to use it and understand the details better, so we’ll get into that now!

## 2.6 What do we do with a model?

Once we have a working model, there are two primary ways we can use it. One way to use a model is to help us understand the relationships between the features and our outcome of interest. In this way, the focus can be said to be on **explanation**, or interpreting the model results. The other way to use a model is to make estimates about the outcome for specific observations, often ones we haven’t seen in our data. In this way the focus is on **prediction**. In practice, we often do both, but the focus is usually on one or the other. We’ll cover both in detail eventually, but let’s start with prediction.

### 2.6.1 Prediction

It may not seem like much at first, but a model is of no use if it can't be used to make predictions about what we can expect in the world around us. Once our model has been *fit* to the data, we can obtain our predictions by plugging in values for the features that we are interested in, and, using the corresponding weights and other parameters that have been estimated, come to a guess about a specific observation. Let's go back to our results, starting with a simpler depiction.

feature	estimate	std_error	statistic	p_value	conf_low	conf_high
intercept	3.49	0.04	82.43	0.00	3.41	3.57
word_count	-0.04	0.00	-11.58	0.00	-0.05	-0.04

The table shows the **coefficient** for each feature including the intercept, which is our starting point. In this case, the coefficient for word count is -0.04, which means that for every additional word in the review, the rating goes down by -0.04 stars. So if we had a review that was 10 words long, we would *predict* a rating of  $3.49 + 10 * -0.04 = 3.1$  stars.

When we're talking about predictions for a linear model, we usually will see this as the following mathematically:

$$\hat{y} = b_0 + b_1x_1 + b_2x_2 + \dots + b_nx_n$$

What is  $\hat{y}$ ? The hat over the  $y$  just means that it's a predicted value of the model, rather than the one we actually observe. Our first equations that just used  $y$  implicitly suggested that we would get a perfect rating value given the model, but that's not the case. We can only get an estimate. The  $\hat{y}$  is also the linear predictor in our graphical version (Figure 2.2), which makes clear it is not the actual target, but a combination of the features that is related to the target.

To make our first equation accurately reflect the relationship between the target and our features, we need to add what is usually referred to as an **error term**,  $\epsilon$ , to account for the fact that our predictions will not be perfect<sup>4</sup>. So the full linear (regression) model is:

$$y = b_0 + b_1x_1 + b_2x_2 + \dots + b_nx_n + \epsilon$$

The error term is a random variable that represents the difference between

---

<sup>4</sup>In most circumstances, if you ever have perfect prediction, or even near perfect prediction, the usual issues are that you have either asked a rather obvious/easy question of your data (e.g., predicting whether an image is of a human or a car), or have accidentally included the target in your features (or a combination of them) in some way.

the actual value and the predicted value, which comes from the weighted combination of features. We can't know what the error term is, but we can estimate it, just like we can the coefficients. We'll talk more about that in the section on estimation (Chapter 4).

TODO: think about which of the following parts of prediction can move to the 'knowing your model' chap

### 2.6.2 What kinds of predictions can we get?

What predictions we get depends on the type of model we are using. For the linear model, we can get predictions for the target, which is a **continuous variable**. Very commonly, we also can get predictions for a **categorical target**, such as whether the rating is 'good' or 'bad'. This simple breakdown pretty much covers everything, as we typically would be predicting a continuous variable or a categorical variable, or more of them, like multiple continuous variables, or a target with multiple categories, or sequences of categories (e.g. words). In our case, we can get predictions for the rating, which is a number between 1 and 5. Had our target been a binary good vs. bad rating, our predictions would still be numeric, and usually expressed as a probability between 0 and 1, say, for the 'good' category. Higher probabilities would mean we'd more likely predict the movie is good. We then would convert that probability to a class of good or bad depending on a chosen probability cutoff. We'll talk about how to get predictions for categorical targets later.

We previously saw a prediction for a single observation, but we can also get predictions for multiple observations at once. In fact, we can get predictions for all observations in our dataset. Besides that, we can also get predictions for observations that we don't even have data for! Fun! The following shows how we can get predictions for all data, and for a single observation with a word count of 5.

#### 2.6.2.1 R

```
all_predictions = predict(model_reviews)

df_prediction = tibble(word_count = 5)
single_prediction = predict(model_reviews, newdata = df_prediction)
```

#### 2.6.2.2 Python

```
all_predictions = model_reviews.predict()

df_prediction = pd.DataFrame({'word_count': [5]})
single_prediction = model_reviews.predict(df_prediction)
```

Here is a plot of our predictions for the observed data versus the actual rat-

ings<sup>5</sup>. The reference line is where the points would fall if we had perfect prediction. We can see that the predictions are definitely not perfect, but we don't expect this. They are not completely off base either, in that generally higher predicted scores are associated with higher observed values. We'll talk about how to assess the quality of our predictions later, but we can at least get a sense that we have a correspondence relationship between our predictions and target, which is definitely better than not having a relationship at all!

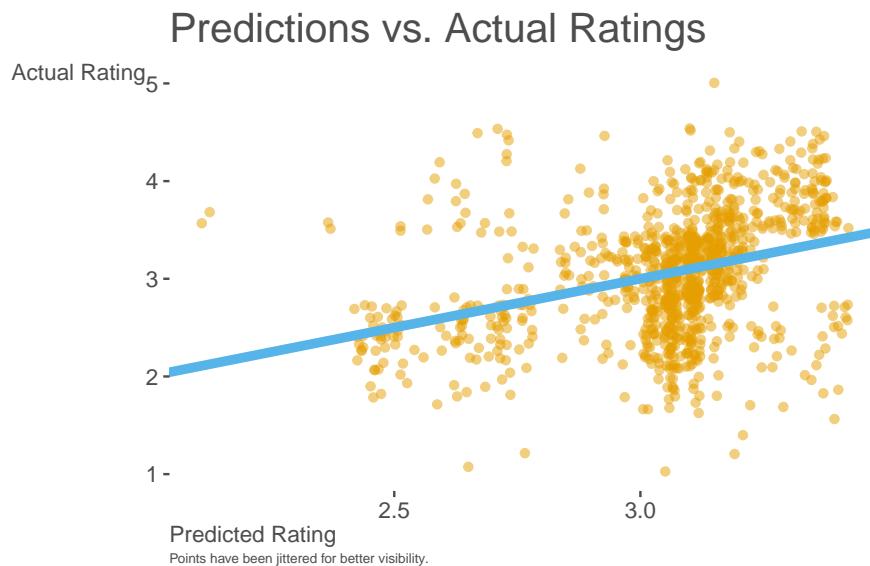


Figure 2.3: Predictions vs. Actual Ratings

Now let's look at what our prediction looks like for a single observation, and we'll add in a few more- one for 10 words, and one for a 50 word review, which is beyond the length of any review in this dataset, and one for 12.3 words, which isn't even possible for this data, since words are only counted as whole values. To get these values we just use the same prediction approach we showed above (`((my-first-model-predictions-r?))`), and we specify the word count value we want to predict for.

Table 2.2: Predictions for Specific Observations

---

<sup>5</sup>Word count is **discrete**- it can only take whole numbers like 3 or 20, and it is our only feature. Because of this, we can only make very limited predicted rating values, while the observed rating can take on many other values. Because of this, the true plot would show a more banded result with many points overlapping, so we use a technique called **jittering** to move the points around a little bit so we can see them all. The points are still roughly in the same place, but they are moved around a little bit so we can see them all.

Word Count	Predicted Rating
5.0	3.3
10.0	3.1
12.3	3.0
50.0	1.4

The values reflect the negative coefficient from our model, reflecting decreasing ratings with increasing word counts. Furthermore, we see the power of the model's ability to make predictions for what we don't see in the data. Maybe we limited our data review size, but we know there are 50 word reviews out there, and we can still make a guess as to what the rating would be for such a review. Maybe in another case, we know a group of people who have on average 12.3 word reviews, and we can make a guess as to what the average rating would be for that group. Our model doesn't know anything about the context of the data, but we can use our knowledge to make predictions about the world around us. This is a very powerful capability, and it's one of the main reasons we use models in the first place.

TODO: summarize some thoughts but move bulk to 'knowing your model' chapter

### 2.6.3 Prediction Error

As we have seen, predictions are not perfect, and an essential part of the modeling endeavor is to better understand these errors and why they occur. In addition, error assessment is the fundamental way in which we assess a model's performance, and, by extension, compare that performance to other models. In general, prediction error is the difference between the actual value and the predicted value or some function of it, and in statistical models, is also often called the **residual**. We can look at these individually, or we can look at them in aggregate with a single metric.

Let's start with looking at the residuals visually. Often the modeling package you use will have this as a default plotting method when doing a standard linear regression, so it's wise to take advantage of it. We plot both the distribution of raw error scores and the cumulative distribution of absolute prediction error. Here we see a couple things. First, the distribution is roughly normal, which is a good thing, since statistical linear regression assumes our error is normally distributed, and the prediction error serves as an estimate of that. Second, we see that the mean of the errors is zero, which is a consequence of linear regression, and the reason we look at other metrics when assessing model performance. We can also see that most of our predictions are within 1 star rating.

Of more practical concern is that we don't see extreme values or clustering,

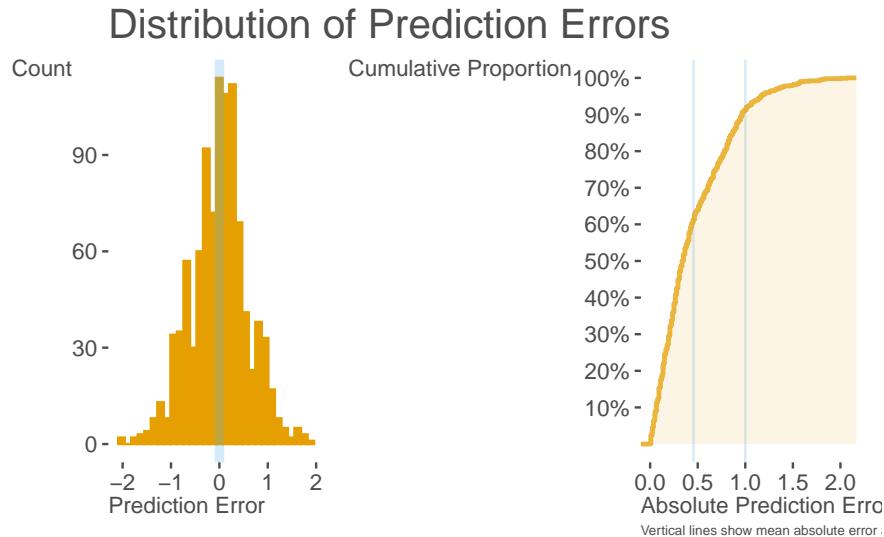


Figure 2.4: Distribution of Prediction Errors

which might indicate a failure on the part of the model to pick up certain segments of the data. It can still be a good idea to look at the extremes just in case we can pick up on some aspect of the data that we could potentially incorporate into the model.

Looking at our worst prediction in absolute terms, we see the observation has a typical word count, and so our simple model will just predict a fairly typical rating. But the actual rating is 1, which is 2.1 away from our prediction, a very noticeable difference. Further data inspection may be required to figure out why this came about.

Table 2.3: Worst Prediction

rating	prediction	word_count
1.0	3.1	10

We can also get an overall assessment of the prediction error. In the case of the linear model we've been looking at, we can express this in a single metric as the sum or mean of our (squared) errors, the latter of which is a very commonly used modeling metric- **MSE** or **mean squared error**, or also, its square root - **RMSE** or **root mean squared error**.

If we look back at our results, we can see this expressed as the part of the

output or as an attribute of the model<sup>6</sup>. The RMSE is more interpretable, as it gives us a sense that our typical errors bounce around by about 0.59. Given that the rating is on a 1-5 scale, this maybe isn't bad, but we could definitely hope to do better than get within roughly half a point on this scale. We'll talk about ways to improve this later.

### 2.6.3.1 R

```
summary(model_reviews) # 'Residual standard error' is approx RMSE

Call:
lm(formula = rating ~ word_count, data = df_reviews)

Residuals:
    Min      1Q  Median      3Q     Max 
-2.0648 -0.3502  0.0206  0.3352  1.8498 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 3.49164   0.04236   82.4   <2e-16 ***
word_count -0.04268   0.00369  -11.6   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.591 on 998 degrees of freedom
Multiple R-squared:  0.118, Adjusted R-squared:  0.118 
F-statistic: 134 on 1 and 998 DF,  p-value: <2e-16
```

### 2.6.3.2 Python

```
np.sqrt(model_reviews.scale) # RMSE
0.590728780660127
```

At this point you have the gist of prediction and prediction error, but there is a lot more to it. More detail can be found in the estimation chapter (Chapter 4), since we often estimate the parameters of our model by picking those that will reduce the prediction error the most. Makes sense right? For now though, let's move on to the other main use of models, which is to help us understand the relationships between the features and the target, or **explanation**.

TODO: This could possibly be moved to 'knowing your model' chapter. Instead, we could just focus on the coefficient/uncertainty/r2 stuff here, and

---

<sup>6</sup>The actual divisor for linear regression output depends on the complexity of the model, and in this case the sum of the squared errors is divided by N-2 (due to estimating the intercept and coefficient) instead of N. This is a technical detail that would only matter for data too small to make much of in the first place, and not important for our purposes here.

then move to the ‘knowing your model’ chapter for the rest. We could also use other models that would better demonstrate SHAP and other metrics.

---

## 2.7 How do we interpret the model?

When it comes to interpreting the results of our model, there are a lot of tools at our disposal, though many of the tools we can ultimately use will depend on the specifics of the model we have employed. In general though, we can group our approach to understanding results at the **feature level** and the **model level**. A feature level understanding regards the relationship between a single feature and the target. Beyond that, we also attempt comparisons of feature contributions to prediction, i.e., relative importance. Model level interpretation is focused on assessments of how well the model ‘fits’ the data, or more generally, predictive performance. We’ll start with the feature level, and then move on to the model level.

### 2.7.1 Feature Level

As mentioned, at the feature level, we are primarily concerned with the relationship between a single feature and the target. More specifically, we are interested in the direction and magnitude of the relationship, but in general, it all boils down to how a feature induces change in the target. For numeric features, we are curious about the change in the target given some amount of change in the feature values. It’s conceptually the same for categorical features, but often we like to express the change in terms of group mean differences or something similar, since the order of categories is not usually meaningful. An important aspect of feature level interpretation is the specific predictions we can get by holding the data at key feature values.

#### 2.7.1.1 Basics

Let’s start with the basics by looking again at our coefficient table from the model output.

feature	estimate	std_error	statistic	p_value	conf_low	conf_high
intercept	3.49	0.04	82.43	0.00	3.41	3.57
word_count	-0.04	0.00	-11.58	0.00	-0.05	-0.04

Here, the main thing to look at are the actual feature coefficients and the direction of their relationship, positive or negative. We saw before that the coefficient for word count is -0.04, and this means that for every additional

word in the review, the rating goes down by -0.04. So if we had a review that was 10 words long, we would predict a rating of  $3.49 + 10 * -0.04 = 3.1$  stars.

This interpretation gives us directional information, but how can we interpret the magnitude of the coefficient? Let's try and use some context to help us. The value for the coefficient is -0.04, and the standard deviation of the rating score, i.e., how much it moves around naturally on its own, is 0.63. So the coefficient is about 6% of the standard deviation of the target. In other words, the addition of a single word to a review results in an expected decrease of 6% of what the review would normally bounce around in value. We might not consider this large, but also, a single word change isn't much. What would be a significant change in word count? Let's consider the standard deviation of the feature. In this case, it's 5.07 for word count. So if we increase the word count by one standard deviation, we expect the rating to decrease by  $-0.04 * 5.07 = -0.2$ . That decrease then translates to a change of  $-0.2 / 0.63 = -0.32$  standard deviation units of the target. Without additional context, many would think that's a significant change<sup>7</sup>, or at the very least, that the coefficient is not negligible, and that the feature is indeed related to the target. But we can also see that the coefficient is not so large that it's not believable.

### 💡 Standardized Coefficients

The calculation we just did results in what's often called a 'standardized' or 'normalized' coefficient. In the case of the simplest model with only one feature like this, it is identical to the Pearson r correlation metric, which we invite you to check and confirm on your own, which should roughly equal our calculation using rounded values. In the case of multiple features, it represents a (partial) correlation between the target and the feature, after adjusting for the other features. But before you start thinking of it as a measure of *importance*, it is not. It provides some measure of the feature-target linear relationship, but that doesn't entail *practical* importance, nor is it useful in the presence of nonlinear relationships, interactions, and a host of other interesting things that are typical to data and models.

After assessing the coefficients, next up in our table is the **standard error**. The standard error is a measure of how much the coefficient varies from sample to sample. If we collected the data multiple times, even under practically identical circumstances, we wouldn't get the same value each time - it would bounce around a bit, and the standard error is an estimate of how much it would bounce around. In other words, the standard error is a measure of **un-**

<sup>7</sup>Historically, people cite Cohen (2009) for effect size guidelines for simple models, but such guidelines are notoriously problematic. Rely on your own knowledge of the data, provide reasons for your conclusions, and let others draw their own. If you cannot tell what would constitute a notable change in your outcome of interest, you probably shouldn't be modeling it in the first place.

**certainty**, and along with the coefficients, it's used to calculate everything else in the table. The statistic, here a t-statistic from the student t distribution<sup>8</sup>, is the ratio of the coefficient to the standard error. This gives us a sense of the effect relative to its variability, but the statistic's primary use is to calculate the **p-value** related to its distribution<sup>9</sup>, which is the probability of seeing a coefficient as large as the one we have, *if* we assume from the outset that the true value of the coefficient is zero. In this case, the p-value is 3.47e-29, which is very small. We can conclude that the coefficient is statistically different from zero, and that the feature is related to the target, at least statistically speaking. However, the interpretation we used regarding the coefficient previously is far more useful than the p-value, as the p-value can be affected by many things not necessarily related to the feature-target relationship, such as sample size, and is often misinterpreted.

Aside from the coefficients, the most important output is the **confidence interval** (CI). The CI is a range of values that encapsulates the uncertainty we have in our guess about the coefficients. While our best guess for the effect of word count on rating is -0.04, we know it's not *exactly* that, and the CI gives us a range of reasonable values we might expect the effect to be based on the data at hand and the model we've employed. In this case, the default is a 95% confidence interval, and we can think of this particular confidence interval like throwing horseshoes<sup>10</sup>. If we kept collecting data and running models, 95% of our CIs would capture the true value, and this is one of the many possible CIs we could have gotten. That's the technical definition, which is a bit abstract<sup>11</sup>, but we can also think of it more simply as a range of values that are good guesses for the true value. In this case, the CI is -0.05 to -0.035, and we can be 95% confident that a good range for the coefficient is between those values. We can also see that the CI is relatively narrow, which is good, as it implies that we have a good idea of what the coefficient is. If it was very wide, we would have a lot of uncertainty about the coefficient, and we would not likely not want to base important decisions regarding it.

---

<sup>8</sup>Most statistical tables of this sort will use a t (student t distribution), Z (normal distribution), or F (F distribution) statistic. It doesn't really matter for your purposes which is used by default, they provide the p-value of interest to claim statistical significance.

<sup>9</sup>You can calculate this as `pt(stat, df = model$degrees_of_freedom, lower=FALSE)*2` in R or `stats.t.cdf` in Python. The model degrees of freedom are provided in the summary output (a.k.a. residual degrees of freedom). `lower=FALSE` and `*2` are to get the two-sided p-value, which is what we want in this case. When it comes to t and Z statistics, anything over 2 is statistically significant by the common standard of a p-value of .05 or less. Note also that even though output will round it to zero, the true p-value can never be zero.

<sup>10</sup>[https://en.wikipedia.org/wiki/Horseshoes\\_\(game\)](https://en.wikipedia.org/wiki/Horseshoes_(game))

<sup>11</sup>The interpretation regarding the CI is even more nuanced than this, but we'll leave that for another time. For now, we'll just say that the CI is a range of values that are good guesses for the true value. Your authors have used frequentist and Bayesian statistics for many years, and we are fine with both of them, because they both work well enough in the real world. Despite where this ranged estimate comes from, the vast majority use CIs in the same way, and they are a useful tool for understanding the uncertainty in our estimates.



Keep in mind that your model has a lot to say about what you'll be able to say at the feature level. As an example, as we get into machine learning models, you won't have as easy a time with coefficients and their confidence intervals. For now we'll stop here, but there is a lot more to the story when it comes to feature level interpretation, and we'll continue to return to the topic. But first, let's take a look at interpreting things in another way.

💡 The confidence interval and p-value will for coefficients in typical statistical linear models will coincide with one another in that, if for a given alpha significance level, if a 1-alpha% CI includes zero, then your p-value will be greater than alpha, and vice versa. This is because the same standard error is used to calculate both. However, the framework of using a CI vs. using the p-value for claiming statistical significance actually came from individuals that were philosophically opposed. Modern day usage of both is a bit of a mess that would upset both Fisher (p-value guy) and Neyman (CI guy), but we'll leave that for another time.

### 2.7.2 Is it a Good Model?

Thus far, we've focused on interpretation at the feature level. But knowing the interpretation of a feature doesn't do you much good if the model itself is poor! In that case, we also need to assess the model as a whole, and as with the feature level, we can go about this in a few ways. Before getting too carried away with asking whether your model is any good or not, you always need to ask yourself *relative to what?* Many models claim top performance under various circumstances, but which are statistically indistinguishable from many other models. So we need to be careful about how we assess our model, and what we compare it to.

First, we can start with the predictions of our model. As noted previously, how well the predictions and target line up is a measure of how well the model fits the data. Most model-level interpretation involves assessing and

comparing model fit and variations on this theme. One of the better ways to assess model fit is visually, so let's look at our predictions vs. the target.

### 2.7.2.1 R

```
predictions = predict(model_reviews)
y = df_reviews$rating
```

### 2.7.2.2 Python

```
predictions = model_reviews.predict()
y = df_reviews.rating
```

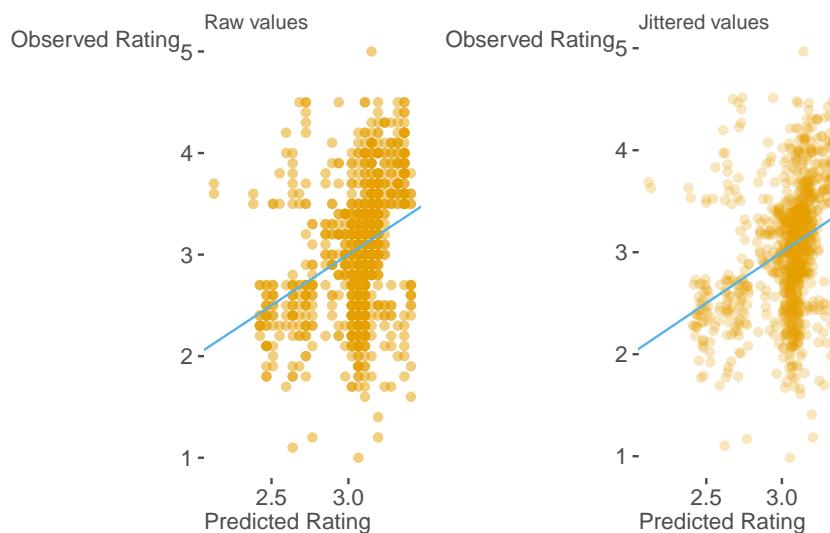


Figure 2.5: Predictions vs. Observed Ratings

The one on the left is using the raw target and predictions, and they appear very grid-like. The reason is that our ratings are only at the single decimal place precision, and our word count is at the integer level precision, so we have a lot of ties. The right side jitters the data randomly a bit so we can see a better pattern, but is otherwise the same. In general, the closer to a line this plot becomes the better, so we can tell already there is still a lot of noise left to explain beyond our model.

### 2.7.2.3 Model Metrics

We've already discussed mean-squared error<sup>12</sup>, but there are other metrics we can use to assess **model fit**. As we noted, (R)MSE is a very popular measure for continuous targets, telling us the standard deviation of errors, or how much they bounce around on average. In our case, the value was 0.59. Another metric we can use in this particular situation is the mean absolute error, which is similar to the mean squared error, but instead of squaring the errors, we just take the absolute value. Conceptually it attempts to get at the same idea, how much our predictions miss on average, and here the value is 0.46, which we actually showed in our initial residual plot (Figure 2.4). With either metric, the closer to zero the better, since as we get closer, we are reducing error.

We can also look at the **R-squared** ( $R^2$ ) value of the model.  $R^2$  is possibly the most popular measure of model performance with linear regression and linear models in general. Before squaring, it's just the correlation of the values that we saw in the previous plot (Figure 2.5). When we square it, we can interpret it as a measure of how much of the variance in the target is explained by the model. In this case, our model shows the  $R^2$  is 0.12, which is not bad for a single feature model in this type of setting. We interpret it that 12% of the target is explained by our model. In addition, we can also interpret  $R^2$  as  $1 - \frac{\text{MSE}}{\text{var}(y)}$ . In other words the complement of  $R^2$  is the proportion of the variance in the target that is not explained by the model. Either way, since ~11% is not explained by the model, our result suggests there is plenty of work left to do!

Note also, that with  $R^2$  we get a sense of the variance shared between *all* features in the model and the target, however complex the model gets. As long as we use it descriptively as a simple correspondence assessment of our predictions and target, it's a fine metric. For various reasons, it's not a great metric for comparing models to each other, but again, as long as you don't get carried away, it's okay to use.

### 2.7.3 Prediction vs. Explanation

In your humble authors' views, one can't stress enough the importance of a model's ability to predict the target. It can be a poor model, maybe because the data is not great, or perhaps we're exploring a new area of research, but we'll always be interested in how well a model **fits** the observed data, and predicts new data.

Even to this day, **statistical significance** is focused on a great deal, even

---

<sup>12</sup>Any time we're talking about MSE, we're also talking about RMSE as it's just the square root of MSE, so which you choose is mostly arbitrary.

to the point that a much hullabaloo is made about models that have no predictive power at all. As strange as it may sound, you can read whole journal articles, news features, and business reports in many fields with hardly any mention of prediction. The focus is almost entirely on the **explanation** of the model, and usually the statistical significance of the features. In those settings, statistical significance is often used as a proxy for importance, which it never should be. Unfortunately, statistical significance is affected by other things besides the size of the coefficient, and without an understanding of the context of the features, in this case, like how long typical reviews are, what their range is, what variability of ratings is, etc., the information it provides is extremely limited, and many would argue, not even useful at all. If we are very interested in the coefficient or weight value specifically, it is better to focus on the range of possible values, which is provided by the confidence interval. While a confidence interval is also a loaded description of a feature's relationship to the target, we can use it in a very practical way as a range of possible values for that weight, and more importantly, *think of possibilities rather than certainties*.

Suffice it to say at this point that how much one focuses on prediction vs. explanation depends on the context and goals of the data endeavor. There are cases where predictive capability is of utmost importance, and we care less about explanatory details, but not to the point of ignoring it. For example, even with deep learning models for image classification, where the inputs are just RGB values, we'd still like to know what the (notably complex) model is picking up on, otherwise we may be classifying images based on something like image backgrounds (e.g. outdoors vs. indoors) instead of the objects of actual interest (dogs vs. cats). In some business or other organizational settings, we are very or even mostly interested in the coefficients/weights, which might indicate how to allocate monetary resources in some fashion. But if those weights come from a model with no predictive power, placing much importance on them may be a fruitless endeavor.

In the end we'll need to balance our efforts to suit the task at hand. Prediction and explanation are both fundamental to the modeling endeavor.

TODO: Move interactions to GLM and shap etc. to 'knowing your model' chapter. This should have multiple features

---

## 2.8 Adding Complexity

We've seen how to fit a model with a single feature and interpret the results, and that helps us to get oriented to the process. However, we'll always have more than one feature for a model except under some very specific circum-

stances, such as exploratory data analysis. So let's see how we can do that with a model that makes more sense.

### 2.8.1 Multiple Features

We can add more features to our model very simply. Using the standard functions we've already demonstrated, we just add them to the formula (both R and statsmodels) as follows.

```
'y ~ feature_1 + feature_2 + feature_3'
```

In other cases, additional features will just be the additional input columns, and nothing about the model code actually changes. We might have a lot of features, and even simpler for linear models this could be dozens in some scenarios. A compact depiction of our model uses the matrix representation, which we'll show in the callout below, and you can find more detail in the matrix section Appendix C overview. For our purposes, all you really need to know is that this:

$$y = X\beta \quad \text{or} \quad y = \alpha + X\beta \quad (2.2)$$

is the same as this:

$$y = \alpha + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 \dots$$

where  $X$  is a matrix of features<sup>13</sup>, and  $\beta$  is a vector of coefficients. Matrix multiplication provides us an efficient way to get our expected value/prediction.

#### Matrix Representation of a Linear Model

Here we'll show the matrix representation form of the linear model, for the typical case where we have more than one feature in the model. In the following,  $y$  is a vector of all target observations, and likewise each  $x$  is a (row) vector of all observations for that feature. The  $b$  vector is the vector of coefficients. The 1 serves as a means to incorporate the intercept. It's just a feature that always has a value of 1. The matrix multiplication is just a compact way of expressing the sum of the features multiplied by their coefficients. We can even do it more

Here is  $y$  as a vector of observations,  $n \times 1$ .

---

<sup>13</sup>For linear regression as we estimate it here, there is actually an additional column at the beginning of the matrix that is all ones, which is a way to incorporate the intercept. However, most models that use a matrix as input will not have the intercept column, as it's not part of the model estimation or is estimated separately.

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \quad (2.3)$$

Here is the vector for  $\mathbf{x}$ , including the intercept:

$$\mathbf{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \dots & x_{1p} \\ 1 & x_{21} & x_{22} & \dots & x_{2p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \dots & x_{np} \end{bmatrix} \quad (2.4)$$

And finally, here is the vector of coefficients:

$$\mathbf{b} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_p \end{bmatrix} \quad (2.5)$$

Putting it all together, we get the linear model in matrix form:

$$\mathbf{y} = \mathbf{X}\mathbf{b} \quad (2.6)$$

You will also see it depicted in a transposed fashion, such that  $y = \beta X$ , which is just a matter of preference, except that it assumes your data is formatted in a way you'll likely never see in practice, where the features are the rows and the observations are the columns. We'll stick with the above representation, which is the more common way you'll have your data organized.

With that in mind, let's get to our model! In what follows, we keep the word count, but now we add some aspects of the reviewer, such as age and the number of children in the household, and features related to the movie, like the release year, the length of the movie in minutes, and the total reviews received. We'll also add another review level feature- the year the review was written. We'll use the same approach as before, and literally just add them as we depicted in our linear model formula (Equation 2.1).

### 2.8.1.1 R

```
model_reviews_extra = lm(
  rating ~
  word_count
  + age
  + review_year
  + release_year
  + length_minutes
```

```

+ children_in_home
+ total_reviews,
data = df_reviews
)

summary(model_reviews_extra)

Call:
lm(formula = rating ~ word_count + age + review_year + release_year +
length_minutes + children_in_home + total_reviews, data = df_reviews)

Residuals:
    Min      1Q  Median      3Q     Max 
-1.8231 -0.3399  0.0107  0.3566  1.5144 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) -4.56e+01   7.46e+00  -6.11  1.5e-09 ***
word_count   -3.03e-02   3.33e-03 -9.10  < 2e-16 ***
age          -1.69e-03   9.24e-04 -1.83   0.0683 .  
review_year   9.88e-03   3.23e-03  3.05   0.0023 ** 
release_year  1.33e-02   1.79e-03  7.43   2.3e-13 ***
length_minutes 1.67e-02   1.53e-03 10.90  < 2e-16 ***
children_in_home 1.03e-01   2.54e-02  4.05   5.5e-05 ***
total_reviews   7.62e-05   6.16e-06 12.36  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.52 on 992 degrees of freedom
Multiple R-squared:  0.321, Adjusted R-squared:  0.316 
F-statistic:  67 on 7 and 992 DF,  p-value: <2e-16

```

### 2.8.1.2 Python

```

model_reviews_extra = smf.ols(
    formula = 'rating ~ word_count \
+ age \
+ review_year \
+ release_year \
+ length_minutes \
+ children_in_home \
+ total_reviews',
    data = df_reviews
).fit()

model_reviews_extra.summary(slim = True)

```

```
<class 'statsmodels.iolib.summary.Summary'>
"""
    OLS Regression Results
=====
Dep. Variable:          rating   R-squared:           0.321
Model:                 OLS      Adj. R-squared:       0.316
No. Observations:      1000     F-statistic:         67.02
Covariance Type:    nonrobust  Prob (F-statistic):  3.73e-79
=====
              coef    std err        t      P>|t|      [0.025    0.975]
-----
Intercept     -45.5688    7.463     -6.106     0.000    -60.215    -30.923
word_count    -0.0303    0.003     -9.102     0.000    -0.037    -0.024
age           -0.0017    0.001     -1.825     0.068    -0.004     0.000
review_year    0.0099    0.003      3.055     0.002     0.004     0.016
release_year   0.0133    0.002      7.434     0.000     0.010     0.017
length_minutes 0.0167    0.002     10.897     0.000     0.014     0.020
children_in_home 0.1028    0.025      4.051     0.000     0.053     0.153
total_reviews  7.616e-05  6.16e-06     12.362     0.000    6.41e-05   8.83e-05
=====
```

Notes:

- [1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
- [2] The condition number is large, 2.82e+06. This might indicate that there are strong multicollinearity or other numerical problems.

There is definitely more to unpack here, but it's important to note that it's just *more* stuff, not *different* stuff. The model-level components are the same in that we still see  $R^2$  etc., although they are all 'better' (higher  $R^2$ , lower error) because we have a more predictive model. Our coefficients look the same also, and we'd interpret them in the same way. Starting with word count, we see that it's still statistically significant, but it has been reduced just slightly from our previous model where it was the only feature (-0.04 vs. -0.03). Why? This suggests that word count has some non-zero correlation, sometimes called **collinearity**, with other features that are also explaining the target to some extent. Our linear model shows the effect of each feature *controlling for other features*, or, *holding other features constant*<sup>14</sup>. Conceptually this means that the effect of word count is the effect of word count *after* we've accounted for the other features in the model. In this case, an increase of a single word results in a -0.03 drop, even after adjusting for the effect of other features.

---

<sup>14</sup>A lot of statisticians and causal modeling folks get very hung up on the terminology here, but we'll leave that to them as we'd like to get on with things. For our purposes, we'll just say that we're interested in the effect of a feature *after* we've accounted for the other features in the model.

Looking at another feature, the addition of a child to the home is associated with 0.1 bump in rating, accounting for the other features.

Thinking about prediction, how would we get a prediction for a movie rating with a review that is 12 words long, written in 2020, by a 30 year old with one child, for a movie that is 100 minutes long, released in 2015, with 10000 total reviews? Exactly the same as we did before (Section 2.6.2)! We just create a data frame with the values we want, and predict accordingly.

#### 2.8.1.3 R

```
predict_observation = tibble(
  word_count = 12,
  age = 30,
  children_in_home = 1,
  review_year = 2020,
  release_year = 2015,
  length_minutes = 100,
  total_reviews = 10000
)

predict(
  model_reviews_extra,
  newdata = predict_observation
)

1
3.26
```

#### 2.8.1.4 Python

```
predict_observation = pd.DataFrame(
  {
    'word_count': 12,
    'age': 30,
    'children_in_home': 1,
    'review_year': 2020,
    'release_year': 2015,
    'length_minutes': 100,
    'total_reviews': 10000
  },
  index = ['new_observation']
)

model_reviews_extra.predict(predict_observation)

new_observation    3.2595
```

```
dtype: float64
```

In our example we're just getting a single prediction, but don't let that hold you back! You can predict an entire data set if you want, and use any values for the features you want. We'll do this explicitly later on, but for now, try getting a prediction for a different set of values.

TODO: MOVE TO KNOWING YOUR MODEL CHAPTER, but maybe mention a bit about added complexity in interpretation

## 2.8.2 More Interpretation Tools

### 2.8.2.1 SHAP Values

Some models are more complicated than can be explained by a simple coefficient, e.g. nonlinear effects in generalized additive models, or may not even have feature-specific coefficients, like gradient boosting models, or may have many parameters associated with a feature, as in deep learning. Such models typically won't come with statistical output like standard errors and confidence intervals either. But we'll still have some tricks up our sleeve to help us figure things out!

A very common interpretation tool is called a **SHAP value**. SHAP stands for **SHapley Additive exPlanations**, and it provides a means to understand how much each feature contributes to a specific prediction. It's based on a concept from game theory called the **Shapley value**, which is a way to understand how much each player contributes to the outcome of a game. The reason we bring it up here is that it has a nice intuition in the linear model case, and demonstrating now is a good way to get a sense of how it works. While the actual computations can be tedious, the basic idea is relatively straightforward- for a given prediction at a specific observation with set feature values, we can calculate the difference between the prediction at that observation and the average prediction. This is the **local effect** of the feature. However, we must also consider doing this for all possible values of other features that might be in a model, as well as considering whether other features are present for the prediction or not. The initial Shapley approach is to average the local effects over all possible combinations of features, which is computationally intractable for all but the simplest model/data settings. The SHAP approach offers more computationally feasible methods for estimation which, while still computationally intensive, is doable for many models. The SHAP approach also has the benefit of being able to be applied to *any* model, and it's the approach we'll use here and return to with some of our other models.

Let's look at the SHAP values for our model. We'll start with a single feature value/observation, using our multifeature model. Here we'll use the first observation where there are 12 words for word count, age of reviewer is 30, a

movie length of 100 minutes etc. To aid our understanding, we calculate the shap value related to word count at that observation by hand, and using a package.

### 2.8.2.2 R

```
# first we need to get the average prediction
avg_pred = mean(predict(model_reviews_extra))

# then we need to get the prediction for the feature value of interest
# for all observations, and average them
pred_observation = predict(
  model_reviews_extra,
  newdata = df_reviews |> mutate(word_count = 12)
)

# then we can calculate the shap value
shap_value_ours = mean(pred_observation) - avg_pred

# we can also use the DALEX package to do this for us
explainer = DALEX::explain(model_reviews_extra, verbose = FALSE)

# observation of interest we want shap values for
obs_of_interest = tibble(
  word_count = 12,
  age = 30,
  children_in_home = 1,
  length_minutes = 100,
  total_reviews = 10000,
  release_year = 2015,
  review_year = 2020,
)

shap_value_package = DALEX::predict_parts_shap(
  explainer,
  obs_of_interest
)

# c(
#   shap_value_ours,
#   shap_value_package['word_count', 'contribution']
# )
```

### 2.8.2.3 Python

```
# first we need to get the average prediction
```

```
avg_pred = model_reviews_extra.predict(df_reviews).mean()

# then we need to get the prediction for the feature value of interest
pred_observation = model_reviews_extra.predict(
    df_reviews.assign(word_count = 12)
)

# then we can calculate the shap value
shap_value_ours = pred_observation.mean() - avg_pred

# now use the shap package for this; it does not work with statsmodels though,
# and single feature models are a bit cumbersome,
# but we still get there in the end!
import shap
from sklearn.linear_model import LinearRegression

# set data up for shap and sklearn
fnames = [
    'word_count',
    'age',
    'review_year',
    'release_year',
    'length_minutes',
    'children_in_home',
    'total_reviews'
]

X = df_reviews[fnames]
y = df_reviews['rating']

# use a linear model that works with shap
model_reviews = LinearRegression().fit(X, y)

# 1000 instances for use as the 'background distribution'
X_sample = shap.maskers.Independent(data = X, max_samples = 1000)

# # compute the SHAP values for the linear model
explainer_linear = shap.Explainer(
    model_reviews.predict,
    X_sample
)

# find an index where word_count is 12
obs_of_interest = pd.DataFrame({
```

```

'word_count': 12,
'age': 30,
'children_in_home': 1,
'review_year': 2020,
'release_year': 2015,
'length_minutes': 100,
'total_reviews': 10000
}, index = ['new_observation'])

shap_values_linear = explainer_linear(obs_of_interest)

shap_value_package = shap_values_linear.values[0, 0]

# (shap_value_ours, shap_value_package)

```

Table 2.4: SHAP Value Comparison

ours	dalex
-0.051	-0.051

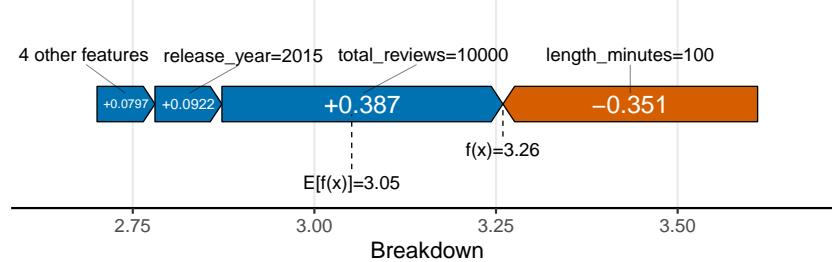
So we see the contribution to a prediction for a single feature, but the shap-related packages provide them for all features, and so we get a sense of each feature's contribution to the prediction. The following shows this as a visualization, as a **force plot** and **waterfall plot**. Smaller contributions are aggregated to one effect to simplify the plot. The dotted line represents the average prediction from our model and the prediction we have for the observation. We see that total reviews and length in minutes contribute most to the prediction at this observation, followed by release year. We can also see that the effect of movie length is negative.

Pretty neat huh? So for any observation we want to inspect, and more importantly, for any model we might use, we can get a sense of how features contribute to that prediction. We also can get a sense of how much each feature contributes to the model as a whole by aggregating these values across all observations in our data, and this provides a measure of **feature importance**, but we'll come back to that in a bit.

If we are concerned with a single feature's relationship with the target, we can also look at the **partial dependence plot** (PDP). The PDP shows the relationship between a feature and the target, but averaged over all other features. In other words, it shows the effect of a feature on the target, but averaged over all other features. For the linear case, it has a direct correspondence to the shap value. The SHAP value is the value the difference between the average prediction and the point on the PDP for a feature at a specific feature value.

We can also look at the **individual conditional expectation** (ICE) plot,

SHAP Force



SHAP Waterfall

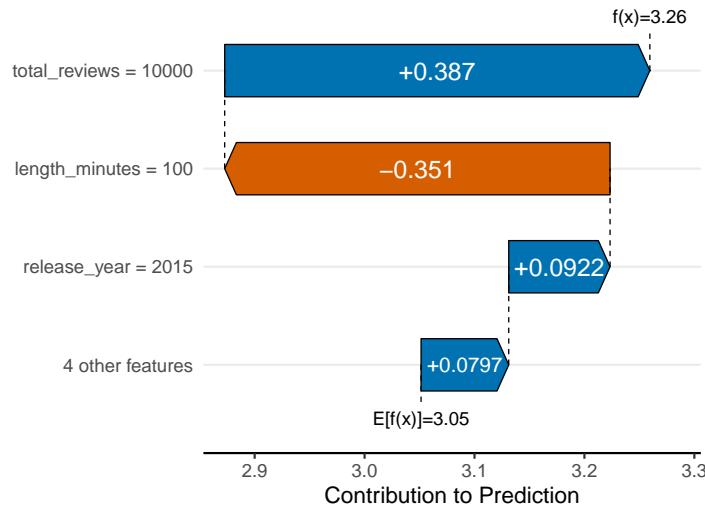


Figure 2.6: SHAP Visualizations

which is a PDP plot for a single observation. By looking at several observations, we can get a sense of the variability in the feature's effect. This becomes more interesting when we have interactions or other nonlinearities in our model.

In addition, there are other plots that are similar to the PDP and ICE, such as the **accumulated local effect** (ALE) plot, which is a bit more robust to correlated features than the PDP plot. Where the PDP and ICE plots show the average effect of a feature on the target, the ALE plot focuses on average *differences* in predictions for the feature at a specific value versus predictions at feature values nearby, and centers the result so that the average difference is zero. We show all three here.

Kinda cool but maybe not so interesting in that they all kind of tell us the same thing about our negative relationship between word count and rating, which we already knew from our coefficient value. The real power will come in later when we use interactions, nonlinear effects, and other models. But it's good to note now that the PDP, ICE, and ALE plots are a nice way to get a sense of the relationship between a feature and the target, and we'll see them again with other models.

### 2.8.3 Feature Importance

How important is a feature? It's a common question, and one that is often asked of models, but the answer ranges from 'it depends' and 'it doesn't matter'. Let's start with some hard facts:

- There is no single definition of importance.
- There is no single metric for *any* model that will definitively tell you how important a feature is relative to others in all data/model contexts.
- There are many metrics for a given model that are equally valid, but may come to different conclusions.
- Any non-zero feature contribution is potentially 'important', however small.
- Many metrics of importance fail to adequately capture interactions and/or deal with correlated features.
- All measures of importance are measured with uncertainty, and the uncertainty can be large.
- Relative to... what? A poor model will still have relatively 'important' features, but they still may not be useful.
- It rarely makes sense to drop features based on importance alone, and doing so will typically drop performance as well.
- In the end, what will you do with the information?

To show just how difficult measuring feature importance is, we only have to stick with our simple linear regression. Think again about  $R^2$ : it tells us the proportion of the target explained by our features. An ideal measure of importance would be able to tell us how much each feature contributes to that

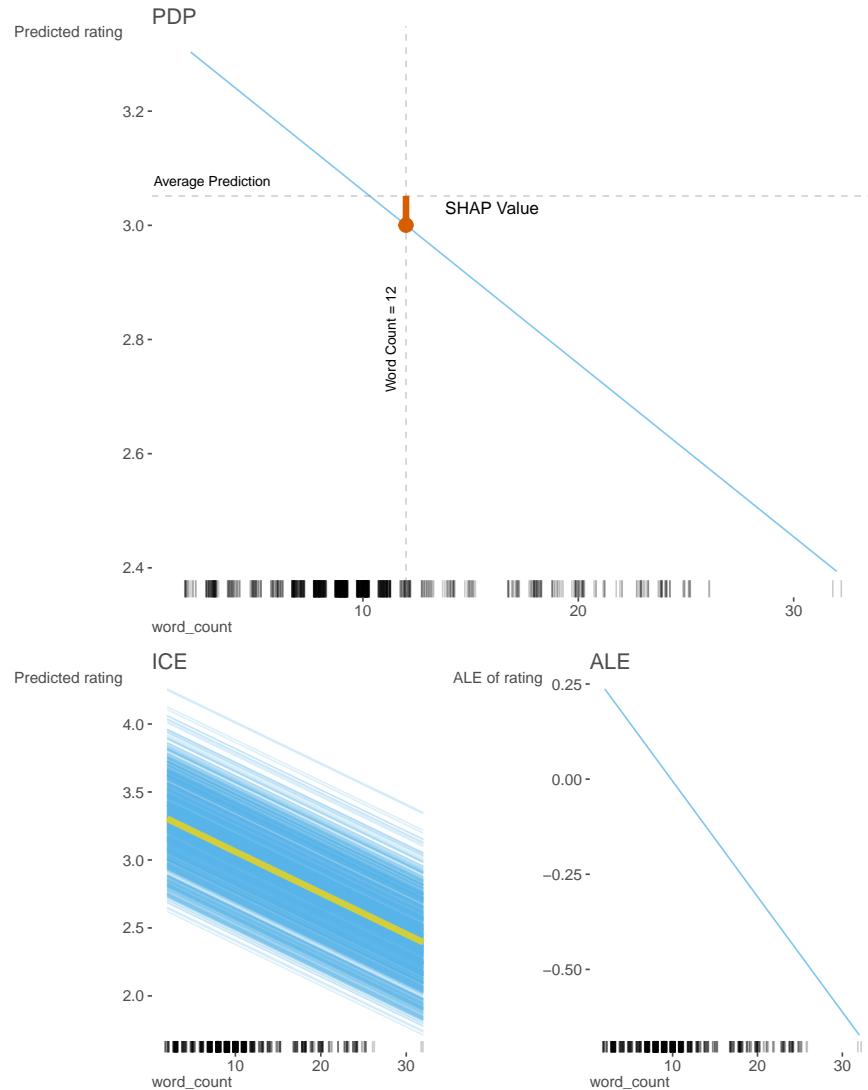


Figure 2.7: PDP, ICE, and ALE Plots

proportion, or in other words, decomposes  $R^2$  into the relative contributions of each feature. One of the most common measures of importance in linear models is the standardized coefficient we demonstrated earlier. You know what it doesn't do? It doesn't decompose  $R^2$  into relative contributions. The easiest situation we could hope for with regard to feature importance is the basic linear model we've been using. Everything is linear, with no interactions, or other things going on. And yet there are many logical ways to determine feature importance, and some even break down  $R^2$  into relative contributions, but they won't necessarily agree with each other in ranking or relative differences. If you can get a measure of statistical difference between whatever metric you choose, it's often the case that 'top' features will not be statistically different from other features. So what do we do? We'll show a few methods here, but the main point is that there is no single answer, and it's important to understand what you're trying to do with the information.

Let's start things off by returning to our SHAP value. If we take the average absolute shap for each feature, we get a sense of the typical contribution size for the features. We can then rank order them as accordingly. Here we see that the most important features here are the number of reviews and the length of the movie. Note that we can't speak to direction here, only magnitude. We can also see that word count is relatively less important.

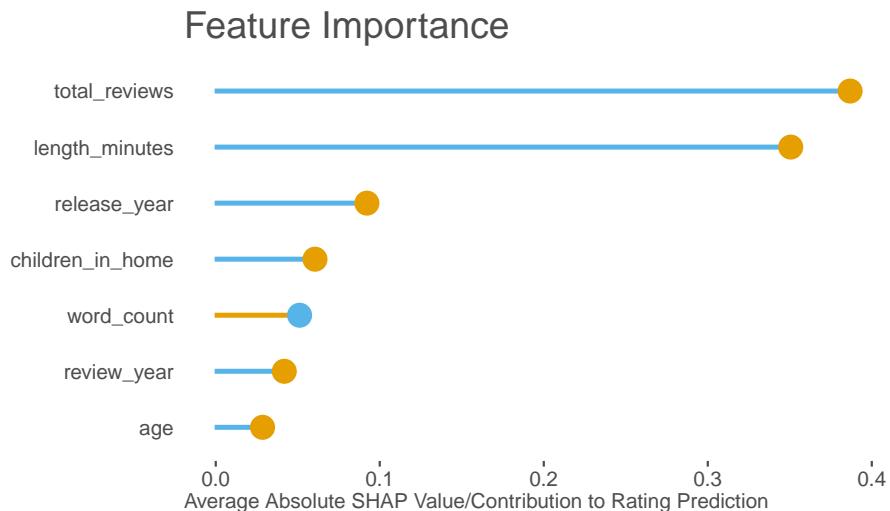


Figure 2.8: SHAP Importance

Now here are some additional methods<sup>15</sup>, some more reasonable than others, some which decompose  $R^2$  and those that do not. Aside from SHAP, the other values represent the proportion of the  $R^2$  value that is attributable to

<sup>15</sup>The non-shap values were provided by the relaimpo package in R.

the feature, or at least attempt to. The ones that truly decompose  $R^2$  are in agreement for the most part and seem to think highly of word count. The others seem to be more varied, and only SHAP devalues word count, but possibly for good reason. Which is best? Which is correct? None. But by looking at a few of these, we can get a sense at least that total reviews and length in minutes are likely useful features to our model.

#### 2.8.4 Model Level Interpretation

As before we can move beyond feature level interpretation, but we are still going to be concerned with the same sorts of questions - how well does the model fit? Have we improved the fit significantly? What do our predictions look like, and so on.

As an example, we see that our  $R^2$  has gone up to 0.32, but this comes with an important caveat - adding any feature would increase our  $R^2$ , even one that was pure noise! So while it is informative, we can also look at MSE or MAE to determine whether the model has improved. In both cases they've been reduced. For example, RMSE is now 0.52, a reduction of 12%, so that provides some confidence that our model has improved over our initial single feature model. There's more we can look at, but at least with we have an idea of how to assess this added complexity at the model level.

##### i Adjusted $R^2$

Part of the output contains an 'adjusted'  $R^2$ . This is a version of  $R^2$  that penalizes the addition of features to the model as a way to account for the fact that adding features will always increase  $R^2$ , even if they are not useful. This is why we can't use  $R^2$  alone to determine whether a model has improved, and why we suggest only considering it as a descriptive statistic. But the adjusted version is kind of a hack, and can even be negative for very poor models. If you want to compare models, use MSE, MAE, and similar metrics.

#### 2.8.5 Other Complexity

TODO: Reconcile with Data chapter, possibly remove from this chapter entirely. If we do, maybe add this model example to that chapter.

##### 2.8.5.1 Categorical Features

Categorical features can be added to a model just like any other feature. The main issue is that they have to be represented numerically, because models only work on numerically coded features and targets. The simplest and most common encoding is called a **one-hot encoding** scheme, which creates a

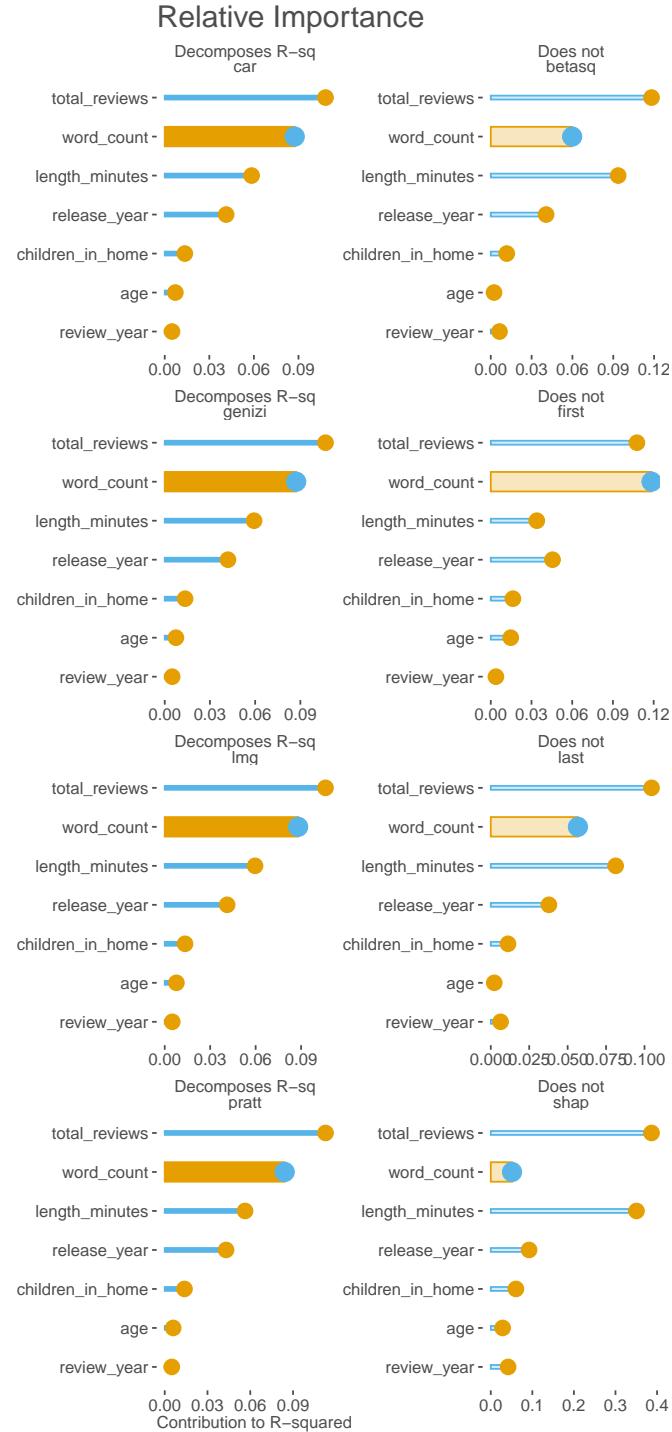


Figure 2.9: Feature Importance by Various Methods

new feature for each category, and assigns a 1 if the observation is in that category, and a 0 otherwise. This is also called a **dummy coding** when used for statistical models. Here is an example of what the coding looks like for the season feature. This is really all there is to it.

rating	season	Fall	Summer	Winter	Spring
2.70	Fall	1	0	0	0
4.20	Fall	1	0	0	0
3.70	Fall	1	0	0	0
2.70	Fall	1	0	0	0
2.40	Summer	0	1	0	0
4.00	Summer	0	1	0	0
1.80	Fall	1	0	0	0
2.40	Summer	0	1	0	0
2.50	Winter	0	0	1	0
4.30	Summer	0	1	0	0

When using statistical models we don't have to do this ourselves. Even other tools for machine learning models will typically have a way to identify and appropriately handle categorical features, even in very complex ways when it comes to deep learning models. What is important is to be aware that they require special handling, but often this is done behind the scenes. Now let's do a quick example using a categorical feature with our data, and we'll keep a numeric feature as well just for consistency.

### 2.8.5.2 R

```
model_cat = lm(
  rating ~ word_count + season,
  data = df_reviews
)

summary(model_cat)

Call:
lm(formula = rating ~ word_count + season, data = df_reviews)

Residuals:
    Min      1Q  Median      3Q     Max 
-1.9184 -0.3622  0.0133  0.3589  1.8372 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)  3.3429    0.0530  63.11 < 2e-16 ***
word_count   -0.0394    0.0036 -10.96 < 2e-16 ***

```

```

seasonSpring -0.0301    0.0622   -0.48     0.63
seasonSummer  0.2743    0.0445    6.17  9.8e-10 ***
seasonWinter -0.0700    0.0595   -1.18     0.24
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.572 on 995 degrees of freedom
Multiple R-squared:  0.176, Adjusted R-squared:  0.173
F-statistic: 53.1 on 4 and 995 DF,  p-value: <2e-16

```

### 2.8.5.3 Python

```

model_cat = smf.ols(
    formula = "rating ~ word_count + season",
    data = df_reviews
).fit()

model_cat.summary(slim = True)

<class 'statsmodels.iolib.summary.Summary'>
"""
=====
          OLS Regression Results
=====
Dep. Variable:           rating   R-squared:      0.176
Model:                 OLS      Adj. R-squared:  0.173
No. Observations:      1000      F-statistic:   53.09
Covariance Type:    nonrobust   Prob (F-statistic):  1.41e-40
=====
              coef    std err         t      P>|t|      [0.025    0.975]
-----
Intercept       3.3429    0.053    63.109      0.000      3.239    3.447
season[T.Spring] -0.0301    0.062   -0.483      0.629     -0.152    0.092
season[T.Summer]  0.2743    0.044    6.171      0.000      0.187    0.362
season[T.Winter] -0.0700    0.059   -1.177      0.239     -0.187    0.047
word_count      -0.0394    0.004   -10.963     0.000     -0.047   -0.032
=====

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
"""

```

We now see the usual output. There is word count again, with its slightly negative association with rating. And we have an effect for each season as well... except, wait a second, where is the fall effect? The coefficients are interpreted the same way - as we move one unit on x, we see a corresponding change in y. But moving from one category to another requires starting at some category

in the first place! So one is chosen arbitrarily, but you would have control over this. In our model, fall is chosen because its first alphabetically. So if we look at say, the effect of summer, we see an increase in the rating of 0.27 relative to fall.

A better approach to understanding categorical features for standard linear models is through what are called **marginal effects**, which can provide a kind of average prediction for each category while accounting for the other features in the model. Better still is to visualize these, and we can use something like our PDP approach from before to do so<sup>16</sup>. It's actually tricky to define 'average' when there are multiple features and interactions involved, so be careful, but we'd interpret the result similarly in those cases as best we can. In this case, we expect higher ratings for summer releases.

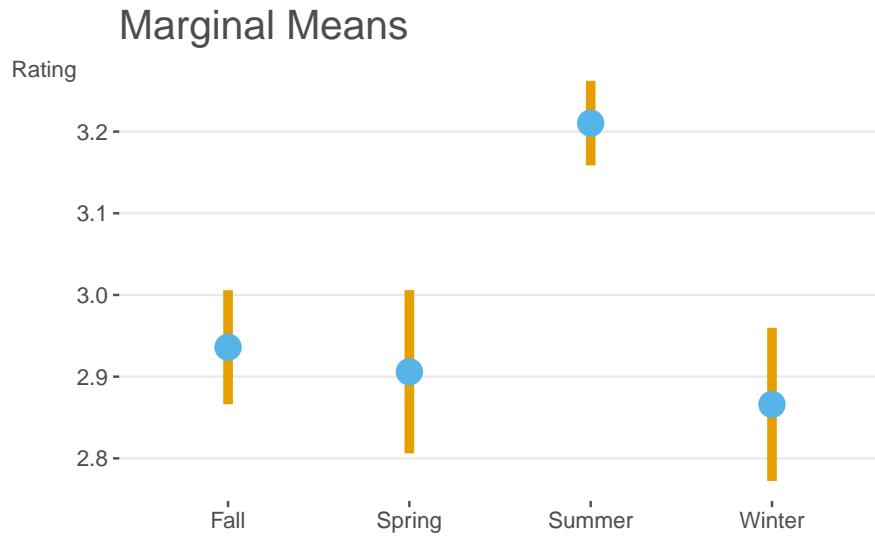


Figure 2.10: Marginal Effects of Season on Rating

TODO: Reconcile with previous section on Interactions

#### 2.8.5.4 Interactions

We'll talk about this more in the other sections, but a common way to add complexity in linear models is through **interactions**. This is where we allow the effect of a feature to vary depending on the values of another feature, or even itself! As a conceptual example, we might expect that the effect of the

---

<sup>16</sup>At the time of this writing, there seems to be very little for this sort of thing in Python. statsmodels provides limited functionality, but only for logistic regression models. In R you have various tools like `marginaleffects`, `emmeans`, `ggeffects` and more.

number of children in the home on rating is different for movies from different genres (much higher for kids movies, maybe lower for horror movies), or that genre and season work together in some way to affect rating (e.g. action movies get higher ratings in summer). We might also consider that the length of a movie might plateau or even have a negative effect on rating after a certain point, i.e., it would have a **curvilinear** effect. All of these are types of interactions we can explore. Interactions allow us to incorporate nonlinear relationships into the model, and so greatly extend the linear model's capabilities - we basically get to use a linear model in a nonlinear way!

TODO: NEED DEMO

---

## 2.9 Assumptions and More

TODO: MOVE BULK OF THIS TO MODEL CRITICISM?? ALSO NEEDS CLEANUP

Every model you use has underlying assumptions which, if not met, could potentially result in incorrect inferences about the effects, performance, or predictive capabilities of the model. The standard linear regression model we've shown is no different, and it has a number of assumptions that must be met for it to be *statistically valid*. Briefly they are:

- That your model is not grossly misspecified (e.g., you've included the right features and not left out important ones)
- The data that you're modeling reflects the population you want to make generalizations about
- The model is linear in the parameters (i.e. no  $\beta_1 \cdot e^x$  type stuff)
- The features are not correlated with the error (prediction errors, unobserved causes)
- Your data observations are independent of each other
- The prediction errors are homoscedastic (don't have large errors with certain predictions vs low with others)
- Normality of the errors (i.e. your prediction errors). Another way to put it is that your target variable is normally distributed conditional on the features.

Things a linear regression model does not assume:

- That the features are normally distributed
  - For example, using categorical features is fine
- That the relationship between the features and target is linear
  - Interactions, polynomial terms, etc. are all fine
- That the features are not correlated with each other

- They usually are

If you do meet these assumptions, it doesn't mean:

- You have large effects
- You have a good model
- You have less uncertainty about your coefficients or predictions

If you don't meet these assumptions, it doesn't mean:

- That your model will have poor predictions
- That your conclusions will necessarily be incorrect

So basically whether or not you meet the assumptions of your model doesn't mean the model is great or terrible, it just means that you have to be careful about what you can say about it. For the linear regression model, if you do meet those assumptions, your coefficient estimates are unbiased<sup>17</sup>, and in general, your statistical inferences are correct ones. If you don't meet them, there are alternative versions of the linear model you could use that would get around the problem. For example, data that runs over a sequence of time (**time series** data) violates the independence assumption since observations closer in time are more likely to be similar than those farther apart. But we would use a **time series** or similar model instead to account for this. If normality is difficult to meet, you could assume a different data generating distribution. We'll discuss some of these in Chapter 6, but it's also important to note that not meeting the assumptions may only mean you'll prefer a different type of linear or other model to use for the data. It's often the case that not meeting the assumptions is often the result of a poor model, e.g., using poor features in an **underspecified** way, like not including interactions or other complexity.

On top of not meeting the assumptions, we may in fact intentionally introduce bias to get better prediction! For example, we might use a **penalized regression** model to reduce the variance in our predictions, at the cost of introducing bias in the coefficients. We'll talk more of this in the Chapter 8, but suffice it to say for now, if you are more interested in prediction, you may be less interested in the statistical assumptions of the basic linear model.

---

<sup>17</sup>This means they are correct on average, not the true value. And if they were biased, this is statistical bias, and has nothing to do with the moral or ethical implications of the data, or whether the features themselves are biased in measurement. Culturally biased data is a different problem than statistical/prediction bias or measurement error, though they are not mutually exclusive. The latter can more readily be tested, while the former is usually more difficult to assess. For example, if our movie reviews only came from a website with a paywall, they would be biased if we wanted to use them to refer to general public opinion. Even then, our model results are perfectly reasonable, as long as they are used to generalize only to that population of people who pay for the website. If we wanted to generalize to the general public, we would need to account for this bias in some way, or use a different data source. MOVE THIS TO SOME OTHER CHAPTER (DATA?)

### 2.9.1 More Complex Models

Let's say you're running some XGBoost or Deep Linear Model and getting outstanding predictions. Assumptions assumptions you say! And you might even be right! But if you want to talk confidently about feature contributions, or know something about the uncertainty in the predictions (which you're assessing right?), well, maybe you might want to know if you're meeting your assumptions. Some of them are:

- You have enough data to make the model generalizable
- Your data isn't biased (e.g., you don't have 90% of your data from one region when you want to talk about a whole area)
- You adequately sampled the hyperparameter space (e.g. you didn't just use the defaults or a small grid search)
- Your observations are independent or at least **exchangeable** and don't have **data leakage**, or you are explicitly modeling observation dependence
- That all the parameter settings you set are correct or at least viable (e.g. you let the model run for a long enough set of iterations, your batch size was adequate, you had enough hidden layers, etc.)

And if you want to talk about specific feature contributions, you are assuming:

- The features are largely uncorrelated
- The features largely do not interact (but then why are you doing a complex model that is inherently interactive), or that your understanding of feature contribution deals with the interactions

Yeah... so, sorry to say, using non-statistical models doesn't mean you don't have to worry about assumptions, you still have some of the old stuff and some new ones to boot.

---

## 2.10 Classification

We've been using a continuous target, but what about a categorical target? For example, what if we just had a binary target of whether a movie was good or bad? We will dive much more into classification models in our upcoming chapters, but it turns out that we can still formulate it as a linear model problem. The main difference is that we use a transformation of our linear combination of features, using what is sometimes called a **link function**, and we'll need to use a different **objective function** rather than least squares, such as the binomial likelihood, to deal with the binary target. This also means we'll move away from  $R^2$  as a measure of model fit, and look at something else, like accuracy.

Graphically we can see it in the following way, which when compared with

our linear model (Figure 2.2), doesn't look much different. In what follows, we create our linear combination and put it through the sigmoid function, which is a common link function for binary targets<sup>18</sup>. The result is a probability, which we can then use to classify the observation as good or bad based on a chosen threshold.

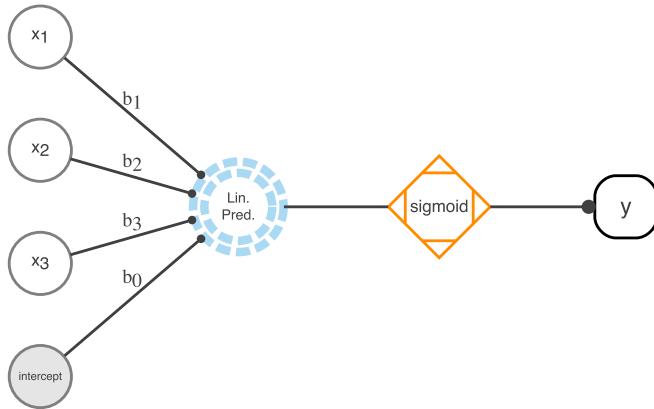


Figure 2.11: A Linear Model with Transformation Can Be a Logistic Regression

As soon as we move away from the standard linear model and use transformations of our linear predictor, simple coefficient interpretation becomes difficult, sometimes exceedingly so. We will explore more of these types of models in Chapter 5.

## 2.11 More linear models

TODO: Minimize this and save/move to final chapter.

Before we leave our humble linear model, let's look at some others. Here is a brief overview of some of the more common linear models you might encounter.

Generalize Linear Models and related

- True GLM e.g. logistic, poisson

---

<sup>18</sup>The sigmoid function in this case is the inverse logistic function, and the resulting statistical model is called logistic regression. In other contexts the model would not be a logistic regression, but this is still a very commonly used *activation function*. But many others could potentially be used e.g. using a normal instead of logistic distribution, resulting in the so-called probit model.

- Other distributions: beta regression, tweedie, t (so-called robust), truncated
- Penalized regression: ridge, lasso, elastic net
- Censored outcomes: Survival models, tobit

Multivariate/multiclass/multipart

- Multivariate regression (multiple targets)
- Multinomial/Categorical/Ordinal regression (>2 classes)
- Zero (or some number) -inflated/hurdle/altered
- Mixture models and Cluster analysis

Random Effects

- Mixed effects models (random intercepts/coefficients)
- Generalized additive models (GAMMs)
- Spatial models (CAR)
- Time series models (ARIMA)
- Factor analysis

Latent Linear Models

- PCA, Factor Analysis
- Mixture models
- Structural Equation Modeling, Graphical models generally

All of these are explicitly linear models or can be framed as such, and most are either identical in description to what you've already seen or require only a tweak or two - e.g. a different distribution, a different link function, penalizing the coefficients, etc. In other cases, we can bounce from one to the another. For example we can reshape our multivariate outcome to be amenable to a mixed model approach, and get the exact same results. We can potentially add a random effect to any model, and that random effect can be based on time, spatial or other considerations. The important thing to know is that the linear model is a very flexible tool that expands easily, and allows you to model most of the types of outcomes we're interested in. As such, it's a very powerful approach to modeling.

---

## 2.12 Wrapping Up

Linear models such as the linear regression demonstrated in this chapter are a very popular tool for data analysis, and for good reason. They are relatively easy to understand, and they are very flexible. They can be used for prediction, explanation, and inference, and they can be used for a wide variety of data types. They are also many tools at our disposal to help us implement and

explore them. But they are not without their limitations, and you'll want to have more in your toolbox than just the approach we've seen so far.

### 2.12.1 Choose your own adventure

Now that you've got the basics, where do you want to go?

- If you want a deeper dive into how we get the results from our model: head to Chapter 4
- If you want to know more about how to understand the model: Chapter 3
- If you want to do some more modeling: Chapter 5, Chapter 6 or Chapter 7
- Got more data questions? Go to the Chapter 10

If you are interested in a deeper dive into the theory and assumptions behind linear models, you can check out more statistical/econometric treatments such as:

- Gelman, Hill, and Vehtari (2020)
- Gelman (2013)
- Harrell (2015)
- Fahrmeir et al. (2013)
- Faraway (2014)
- Wooldridge (2012)
- Greene (2017)
- Gromlund (2023)
- Kuhn and Silge (2023)

But there are many, many books on statistical analysis, linear models, and linear regression specifically. There are those that show nothing but applied results, and those that are very theoretical. Texts tend to get more mathy and theoretical as you go back in time, to the mostly applied and code-based treatments today. You will likely need to do a bit of exploration to find one you like best.

---

## 2.13 Exercise

- Import some data. Stick with the current data if you want and just try out other features, or maybe try the world happiness data 2018 data. You can find details about it in the appendix Appendix B.

TODO: ADD DATA LINK

- Fit a linear model, try to keep it to no more than three features.
- Get all the predictions for the data, and try at least
- Interpret the coefficients

- Assess the model fit
- 

## **2.14 refs**

Molnar (2023)



# 3

---

## *Knowing Your Model*

---

In addition to giving the world one of the greatest television show theme songs – Quincy Jones’ *The Streetbeater – Sanford & Son* gave us an insightful quote for offering criticism: “You big dummy.” While we don’t advocate for swearing at or denigrating your model, how do you know if your model is performing up to your expectations? It is easy to look at your coefficients, *t*-values, and an adjusted  $R^2$ , and say, “Wow! Look at this great model!” Your friends will be envious of such terrific *p*-values, and all of the strangers that you see at social functions will be impressed. What happens if that model falls apart on new data, though? What if a stakeholder wants to know exactly how a prediction was made for a specific business decision? Sadly, all of the stars that you gleefully pointed towards in your console will not offer you any real answers.

Instead of falling in immediate love with your model, you should ask real questions of it. How does it perform on different slices of data? Do predictions make sense? Is your classification cut-point appropriate? In other words, you should criticize your model before you decide it can be used for its intended purposes. Remember that it is **data modeling**, not **data truthing**. In other words, you should always be prepared to call your model a “big dummy”.

---

### 3.1 Key Ideas

- Metrics can help you assess how well your model is performing, and they can also help you compare different models.
- Different metrics can be used depending on the goals of your model.
- Visualizations can help you understand how your model is making predictions and which variables are important.

#### 3.1.1 Why this matters

It’s never good enough to simply get model results. You need to know how well your model is performing and how it is making predictions. You also should be comparing your model to other alternatives. Doing so provides more

confidence in your model and helps you to understand how it is working, and just as importantly, where it fails. This is actionable knowledge.

### 3.1.2 Good to know

This takes some of the things we see in other chapters on linear models and machine learning. We'd suggest have linear model basics down pretty well.

---

## 3.2 Model Metrics

Regression and classification have very different metrics for assessing model performance. We want to give you a sample of some of the more common one, but we also want to acknowledge that there are many more that you can use! We would always recommend looking at a few different metrics to get a better sense of how your model is performing.

### 3.2.1 Regression Metrics

Recall that a primary goal of our standard linear model is to produce  $\hat{y}$  – the predicted outcome. Since we are predicting a value, we need to be able to compare that prediction to its actual value. The closer our prediction is to the actual value, the better our model is performing.

Before we create a model, we are going to read in our data and then create two different splits within our data: a **training** set and a **testing** set. In other words, we are going to **partition** our data so that we can train a model and then see how well that model does with new data.

This basic split is the foundation of **cross-validation**. Cross-validation is a method for partitioning data into training and testing sets, but it does so in a way that allows you to train and test your model multiple times. We will not be covering cross-validation in this book, but we would strongly encourage you to learn more about it.

#### 3.2.1.1 R

```
reviews = read.csv(  
  "data/movie_reviews_processed.csv"  
)  
  
initial_split = sample(  
  x = 1:nrow(reviews),  
  size = nrow(reviews) * .75,
```

```

    replace = FALSE
)

training_data = reviews[initial_split, ]

testing_data = reviews[-initial_split, ]

```

### 3.2.1.2 Python

```

import pandas as pd
import numpy as np

reviews = pd.read_csv("data/movie_reviews_processed.csv")

initial_split = np.random.choice(
    reviews.index,
    size = int(reviews.shape[0] * .75),
    replace = False
)

```

```

training_data = reviews.iloc[initial_split, :]
testing_data = reviews.iloc[-initial_split, :]

```

You'll notice that we created training data with 75% of our data and we will use the other 25% to test our model. With training data in hand, let's produce a model to predict rating

### 3.2.1.3 R

```

model_train = lm(
  rating ~
  review_year_0 + release_year_0 +
  age_sc + length_minutes_sc +
  total_reviews_sc + word_count_sc +
  genre + gender +
  reviewer_type + work_status +
  season,
  training_data
)

```

### 3.2.1.4 Python

```

import statsmodels.api as sm

features = ["review_year_0", "release_year_0",
           "age_sc", "length_minutes_sc",

```

```

"total_reviews_sc", "word_count_sc",
"genre", "gender",
"reviewer_type", "work_status",
"season"]

X_features = training_data[features]
X_features = sm.add_constant(X_features)
X_features = pd.get_dummies(X_features)
X_features = X_features.drop(
    columns=["work_status_Unemployed", "season_Winter"]
)
X_features = X_features.values.astype(float)
y_target = training_data["rating"].values.astype(float)

model_train = sm.OLS(y_target, X_features).fit()

```

Now that we have a model on our training data, we can use it to make predictions on our test data:

### 3.2.1.5 R

```
predictions = predict(model_train, newdata = testing_data)
```

### 3.2.1.6 Python

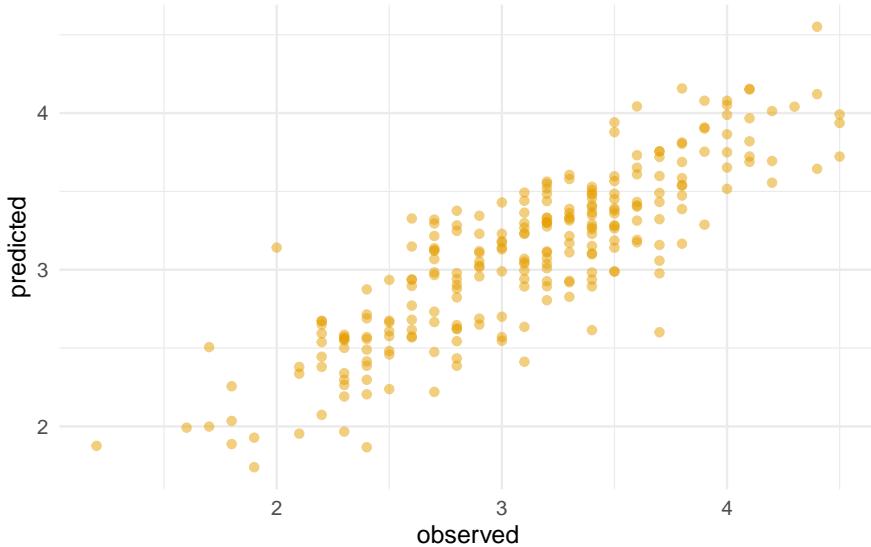
```

X_features_testing = testing_data[features]
X_features_testing = sm.add_constant(X_features_testing)
X_features_testing = pd.get_dummies(X_features_testing)
X_features_testing = X_features_testing.drop(
    columns=["work_status_Unemployed", "season_Winter"]
)
X_features_testing = X_features_testing.values.astype(float)
y_target_testing = testing_data["rating"].values.astype(float)

predictions = model_train.predict(X_features_testing)

```

The goal now is to find out how close our predictions match reality. Let's look at them first:



Obviously, our points do not make a perfect line. Therefore, we need to determine how far off we are. There are a number of metrics that can be used to measure this. We'll go through a few of them here.

#### 3.2.1.7 Mean Squared Error

One of the most common metrics is the mean squared error (MSE). The MSE is the average of the squared differences between the predicted and actual values. It is calculated as follows:

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

MSE is a great metric for penalizing large errors. Since errors are squared, the larger the error, the larger the penalty.

#### 3.2.1.8 R

```
mean((testing_data$rating - predictions)^2)
[1] 0.1040754
Metrics::mse(testing_data$rating, predictions)
[1] 0.1040754
```

#### 3.2.1.9 Python

```
from sklearn.metrics import mean_squared_error
```

```

np.mean((testing_data.rating - predictions)**2)
0.1069698923176642
mean_squared_error(testing_data.rating, predictions)
0.1069698923176642

```

### 3.2.1.10 Mean Absolute Error

The mean absolute error (MAE) is the average of the absolute differences between the predicted and actual values. It is calculated as follows:

$$MAE = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

MAE is a great metric when all you really want to know is how far off your predictions are from the actual values. It is not as sensitive to large errors as the MSE.

### 3.2.1.11 R

```

mean(abs(testing_data$rating - predictions))
[1] 0.2542151
Metrics::mae(testing_data$rating, predictions)
[1] 0.2542151

```

### 3.2.1.12 Python

```

from sklearn.metrics import mean_absolute_error

np.mean(abs(testing_data.rating - predictions))
0.25884294424715865
mean_absolute_error(testing_data.rating, predictions)
0.25884294424715865

```

### 3.2.1.13 Root Mean Squared Error

Perhaps the regression metric that you are most likely to encounter in the wild, the root mean squared error (RMSE) is the square root of the average of the squared differences between the predicted and actual values. It is calculated as follows:

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$$

### 3.2.1.14 R

```
sqrt(mean((testing_data$rating - predictions)^2))
[1] 0.3226072

Metrics::rmse(testing_data$rating, predictions)
[1] 0.3226072
```

### 3.2.1.15 Python

```
from sklearn.metrics import mean_squared_error

np.sqrt(np.mean((testing_data.rating - predictions)**2))
0.32706252050283013

np.sqrt(mean_squared_error(testing_data.rating, predictions))
0.32706252050283013
```

### 3.2.1.16 Mean Absolute Percentage Error

The mean absolute percentage error (MAPE) is the average of the absolute differences between the predicted and actual values, expressed as a percentage of the actual values. It is calculated as follows:

$$MAPE = \frac{1}{n} \sum_{i=1}^n \frac{|y_i - \hat{y}_i|}{y_i}$$

### 3.2.1.17 R

```
mean(
  abs(testing_data$rating - predictions) /
  testing_data$rating
)
[1] 0.08701583

Metrics::mape(testing_data$rating, predictions)
[1] 0.08701583
```

### 3.2.1.18 Python

```
from sklearn.metrics import mean_absolute_percentage_error

np.mean(
    abs(testing_data.rating - predictions) /
    testing_data.rating
)
0.08903596657668966

mean_absolute_percentage_error(testing_data.rating, predictions)
0.08903596657668966
```

### 3.2.1.19 Which To Use?

In the end, it won't hurt to look at a few of these metrics to get a better idea of how well your model is performing. You will **always** be using these metrics to compare different models, so use a few of them to get a better sense of how well your models are performing relative to one another. Does adding a variable help drive down RMSE, indicating that the variable helps to reduce large errors? In other words, does adding complexity to your model provide a big reduction in error? If adding variables doesn't help reduce error, do you really need to include it in your model?

## 3.2.2 Classification Metrics

Whenever we are classifying outcomes, we don't have the same ability to compare a predicted score to an observed score – instead, we are going to use the predicted probability of an outcome, establish a cut-point for that probability, convert everything below that cut-point to 0, and then convert everything at or above that cut-point to 1. We can then compare a table predicted and actual **classes**.

Let's start with a model to predict whether a review is "good" or "bad". We will use the same training and testing data that we created above.

### 3.2.2.1 R

```
logistic_model_train = glm(
  rating_good ~
  review_year_0 + release_year_0 +
  age_sc + length_minutes_sc +
  total_reviews_sc + word_count_sc +
  genre + gender +
  reviewer_type + work_status +
  season,
```

```
    training_data,
    family = binomial
)
```

### 3.2.2.2 Python

```
import statsmodels.api as sm
from statsmodels.genmod.families import Binomial

logistic_model_train = sm.GLM(
    training_data.rating_good,
    X_features,
    family = Binomial()
).fit()
```

Now that we have our model trained, we can use it to get the predicted probabilities for each observation.

### 3.2.2.3 R

```
predictions = predict(logistic_model_train,
                      newdata = testing_data,
                      type = "response")
```

### 3.2.2.4 Python

```
predictions = logistic_model_train.predict(X_features_testing)
```

We are going to take those probability values and make a decision to convert everything above .49 to the positive class (a “good” review). It is a bold assumption, but one that we will make at first!

### 3.2.2.5 R

```
predictions = ifelse(predictions > .49 , 1, 0)
```

### 3.2.2.6 Python

```
predictions = np.where(predictions > .49, 1, 0)
```

```
predictions = pd.Series(predictions)
```

### 3.2.2.7 Confusion Matrix

The confusion matrix is a table that shows the number of correct and incorrect predictions made by the model.

Let’s give some names to each element in that table, so that we have a little more clarity about what they signify:

0	1
0 TN:73 FN:17	
1 FP:25 TP:135	

- **TN:** A True Negative is an outcome where the model correctly predicts the negative class – the model correctly predicted that the review was not good.
- **FN:** A False Negative is an outcome where the model incorrectly predicts the negative class – the model incorrectly predicted that the review was not good.
- **FP:** A False Positive is an outcome where the model incorrectly predicts the positive class – the model incorrectly predicted that the review was good.
- **TP:** A True Positive is an outcome where the model correctly predicts the positive class – the model correctly predicted that the review was good.

In an ideal world, we would have all of our observations fitting nicely in the diagonal of that table. Unfortunately, we don't live in the ideal world and we always have values in the off diagonal. The more values we have in the off diagonal (i.e., in the FN and FP spots), the worse our model is at classifying outcomes.

Let's look at some metrics that will help to see if we've got a suitable model or not.

### 3.2.2.8 Accuracy

Accuracy is the first thing you see and the last thing that you trust! Of all the metrics to assess the quality of classification, accuracy is the easiest to cheat. If you have any **class imbalance** (i.e., one class within the target has far more observations than the other), you can get a high accuracy by simply predicting the majority class all of the time.

Accuracy's allure is in its simplicity. The accuracy is the proportion of correct predictions made by the model. It is calculated as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

From our table above, we can calculate the accuracy as follows:

### 3.2.2.9 R

```
TN = 88
TP = 123
FN = 10
FP = 29
```

```
(TN + TP) / (TN + TP + FN + FP)
[1] 0.844
```

### 3.2.2.10 Python

```
TN = 88
TP = 123
FN = 10
FP = 29
```

```
(TN + TP) / (TN + TP + FN + FP)
0.844
```

To get around the false sense of confidence that accuracy alone can promote, we can look at a few other metrics.

**⚠️** Seriously, accuracy alone should not be trusted unless you have a perfectly even split in the target! If you find yourself in a meeting where people are presenting their classification models and they only talk about accuracy, you should be very skeptical of their model; this is especially true when those accuracy values seem too good to be true.

### 3.2.2.11 Sensitivity/Recall/True Positive Rate

Sensitivity, also known as recall or the true positive rate, is the proportion of **actual positives** that are correctly identified as such. If you want to know how well your model predicts the positive class, sensitivity is the metric for you. It is calculated as follows:

$$\text{Sensitivity} = \frac{TP}{TP + FN}$$

### 3.2.2.12 R

```
TP / (TP + FN)
[1] 0.924812
```

### 3.2.2.13 Python

```
TP / (TP + FN)
0.924812030075188
```

### 3.2.2.14 Specificity/True Negative Rate

Specificity, also known as the true negative rate, is the proportion of **actual negatives** that are correctly identified as such. If you want to know how well your model will work with the negative class, specificity is a great metric. It is calculated as follows:

$$\text{Specificity} = \frac{TN}{TN + FP}$$

### 3.2.2.15 R

```
TN / (TN + FP)
[1] 0.7521368
```

### 3.2.2.16 Python

```
TN / (TN + FP)
0.7521367521367521
```

### 3.2.2.17 Precision/Positive Predictive Value

The precision is the proportion of **positive predictions** that are correct. It is calculated as follows:

$$\text{Precision} = \frac{TP}{TP + FP}$$

### 3.2.2.18 R

```
TP / (TP + FP)
[1] 0.8092105
```

### 3.2.2.19 Python

```
TP / (TP + FP)
0.8092105263157895
```

### 3.2.2.20 Negative Predictive Value

The negative predictive value is the proportion of **negative predictions** that are correct. It is calculated as follows:

$$NPV = \frac{TN}{TN + FN}$$

### 3.2.2.21 R

```
TN / (TN + FN)
[1] 0.8979592
```

### 3.2.2.22 Python

```
TN / (TN + FN)
0.8979591836734694
```

We can get almost all of that with the `confusionMatrix` function from the `caret` package in R:

```
caret::confusionMatrix(as.factor(predictions),
                      as.factor(testing_data$rating_good),
                      positive = "1")

Confusion Matrix and Statistics

          Reference
Prediction    0    1
      0   73   17
      1   25  135

Accuracy : 0.832
95% CI : (0.7798, 0.8762)
No Information Rate : 0.608
P-Value [Acc > NIR] : 1.287e-14

Kappa : 0.6424

McNemar's Test P-Value : 0.2801

Sensitivity : 0.8882
Specificity : 0.7449
Pos Pred Value : 0.8438
Neg Pred Value : 0.8111
Prevalence : 0.6080
Detection Rate : 0.5400
Detection Prevalence : 0.6400
Balanced Accuracy : 0.8165

'Positive' Class : 1
```

We also get:

- kappa: A measure of how much better the model is than random guessing.  
It is calculated as follows:

$$\kappa = \frac{Accuracy - ExpectedAccuracy}{1 - ExpectedAccuracy}$$

where the expected accuracy is calculated as follows:

$$ExpectedAccuracy = \frac{(TP + FN)(TP + FP) + (FP + TN)(FN + TN)}{(TP + TN + FP + FN)^2}$$

- Prevalence: The proportion of actual positives in the data. It is calculated as follows:

$$Prevalence = \frac{TP + FN}{TP + TN + FP + FN}$$

- Balanced Accuracy: The average of the sensitivity (TPR) and specificity (TNR). It is calculated as follows:

$$BalancedAccuracy = \frac{Sensitivity + Specificity}{2}$$

### 3.2.2.23 Ideal Decision Points

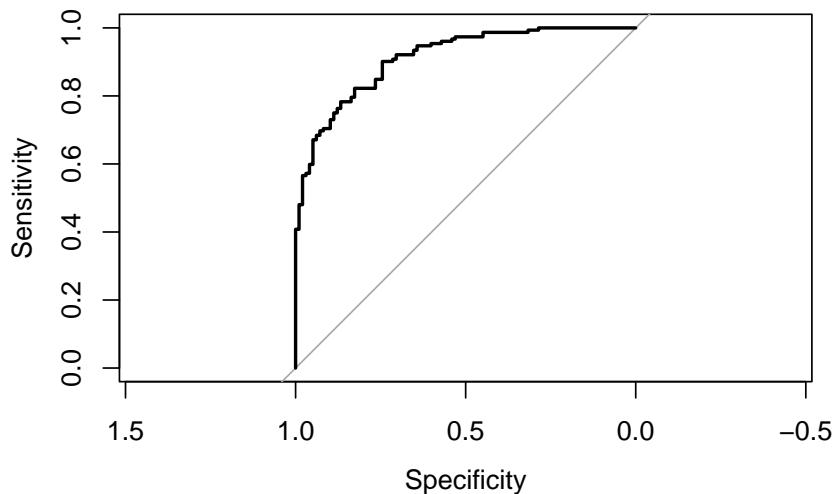
Earlier, we used a predicted probability value of 0.49 to establish our predicted class. That is a pretty bold assumption on our part and we should probably make sure that the cut-off value we choose is going to offer us the best performance.

To handle this task, we will start by creating a **Receiver Operating Characteristic** (ROC) curve. This curve plots the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. The **area under the curve** (AUC) is a measure of how well the model is able to distinguish between the two classes. The closer the AUC is to 1, the better the model is at distinguishing between the two classes.

3.2.2.24 R

```
library(pROC)
```

```
testing_data$rating_good,  
prediction_prob  
)  
  
plot(roc)
```



```
auc(roc)  
  
Area under the curve: 0.9118
```

### 3.2.2.25 Python

```
from sklearn.metrics import roc_curve, auc  
  
fpr, tpr, thresholds = roc_curve(  
    testing_data.rating_good,  
    predictions  
)  
  
auc(fpr, tpr)  
0.8420473607389496
```

With ROC curves and AUC values, we can get a sense of how well our model is able to distinguish between the two classes. Now we can find the ideal cut-point for balancing the TPR and FPR.

### 3.2.2.26 R

```
coords(roc, "best", ret = "threshold", transpose = TRUE)

threshold
0.6904515
```

### 3.2.2.27 Python

```
thresholds[np.argmax(tpr - fpr)]

1
```

Those coordinates are going to give us the “best” decision cut-point. Instead of being naive about setting our probability to .5, this will give a cut-point that will lead to better classifications for our testing data.

We will leave it to you to take that ideal cut-point value and update your metrics to see how much of a difference it will make.

Whether it is a meager, modest, or meaningful improvement is going to vary from situation to situation, as will how you determine if your model is “good” or “bad”. If we look back to our original `Balanced Accuracy` value of 0.6490, we’d imagine that our model gets a True Positive or True Negative about 65% of the time, leaving us wrong 35% of the time. Is that good or bad?

## 3.3 Model Visualizations

Using various fit metrics to assess your model’s performance is critical for knowing how well it will do with new data. As good as it might be to know if it is useful, you might also want to know what is actually happening in the model. Which variables are important? How did a specific observation reach its predicted value?

For these tasks, and many others, we can turn to visualizations to gain a better understanding of our model. Afterall, we can’t really criticize something we don’t understand, can we? To help us along, we are going to use `DALEX` to create **model explainers**.

We will focus on two types of explainers: **variable importance** and **localized predictions**. We will look at them individually for regression and classification tasks.

### 3.3.1 Regression

We are going to add some more features to our model to make it a little more interesting, check that model's performance, and then look at the **Partial Dependence Plots**, which will give us a good idea about the relationship between the features and the target.

#### 3.3.1.1 R

```
library(DALEX)

features = c(
  "review_year_0", "release_year_0",
  "age_sc", "length_minutes_sc",
  "total_reviews_sc", "word_count_sc",
  "genre", "gender",
  "reviewer_type", "work_status",
  "season")

train_explain = explain(
  model_train,
  data = training_data[, features],
  y = training_data$rating,
  verbose = FALSE
)

train_performance = model_performance(train_explain)

# train_performance # not shown

train_var_effect = model_profile(train_explain, features)
plot(train_var_effect)
```

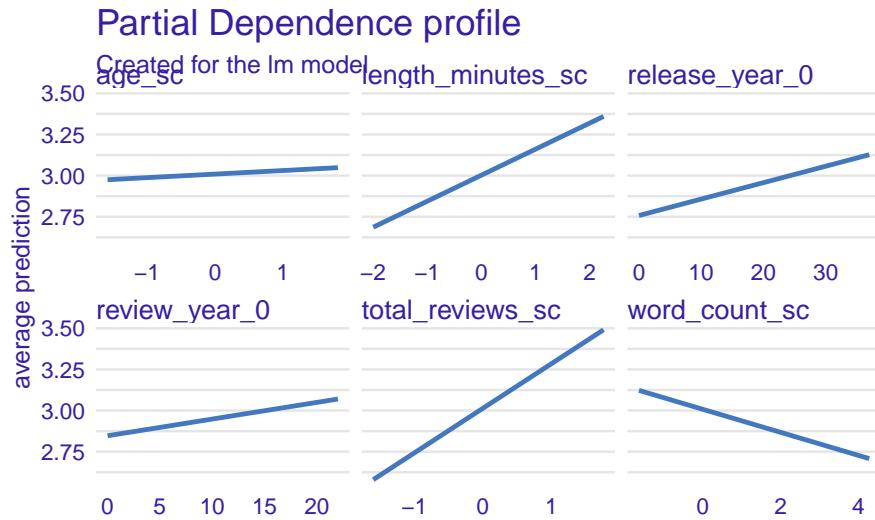


Figure 3.1: R Performance Plot

### 3.3.1.2 Python

```

import dalex as dx
import matplotlib.pyplot as plt

train_explain = dx.Explainer(
    model_train,
    data = X_features,
    y = training_data.rating,
    verbose = False
)

train_performance = train_explain.model_performance()

perf_plot = train_performance.plot()
perf_plot.show()

```

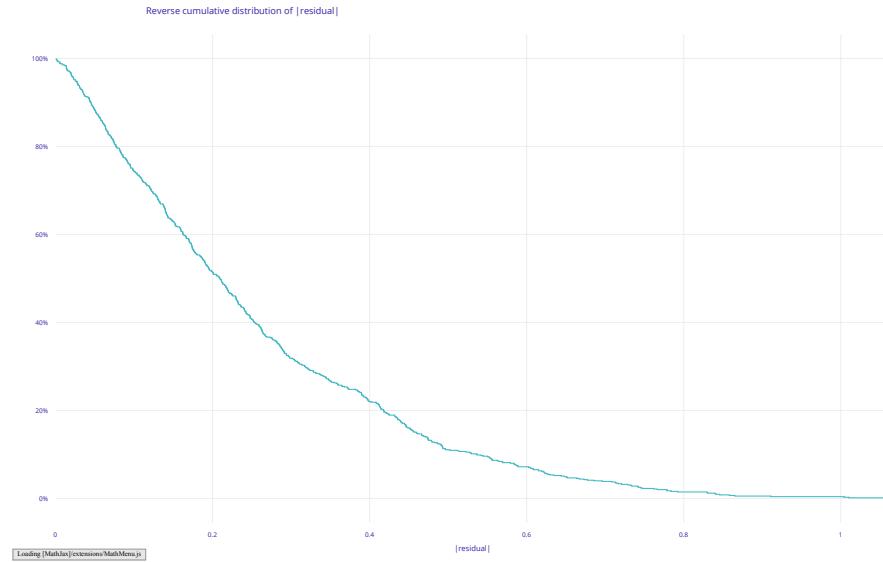


Figure 3.2: Python Performance Plot

We can see what R and Python offer us for model performance plots in Figure 3.1 and Figure 3.2. We can dig into more specific information about our model, beyond just the general performance.

### 3.3.1.3 Variable Importance

As with any model, knowing which variables are important is a critical piece of information. We can use the `model_parts` function to get a sense of which variables are most important to our model. Dalex creates feature importance by assessing how a model's RMSE changes when a feature is permuted. The more the loss changes, the more important the feature!

### 3.3.1.4 R

```
model_var_imp = model_parts(
  train_explain, type = "variable_importance"
)
plot(model_var_imp)
```

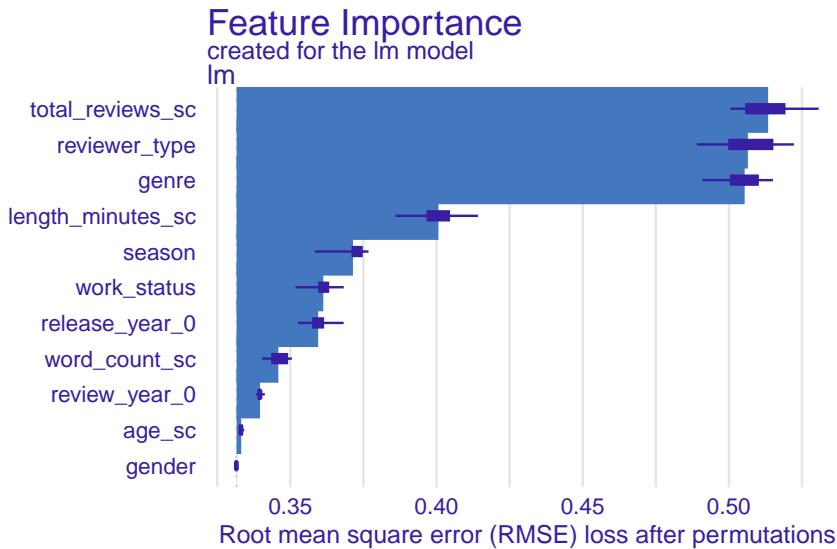


Figure 3.3: R Variable Importance Plot

### 3.3.1.5 Python

```
model_var_imp = train_explain.model_parts(
    type = "variable_importance"
)

model_var_imp.plot()
```

TODO: ONLY SHOW ONE PRETTY PLOT, BUT ALLOW THE CODE TO SHOW HOW TO OBTAIN

In Figure 3.3 and `?@fig-py_var-imp`, we see that `total_reviews_sc`, `length_minutes_sc`, `word_count_sc`, and `release_year_0` are the most important features in our model. Now that we know the variables that are pulling the most weight, we can turn to exploring predictions.

### 3.3.1.6 Localized Predictions

If you are every curious to see how a particular observation reached its predicted value, you can use the `predict_parts` function to get a sense of how each feature contributed to the final prediction. We will look at the second observation in our `testing_data` to see how it was predicted.

### 3.3.1.7 R

```
break_down_plot = predict_parts(
    train_explain,
```

```

new_observation = testing_data[2, ],
type = "break_down")

plot(break_down_plot)

```

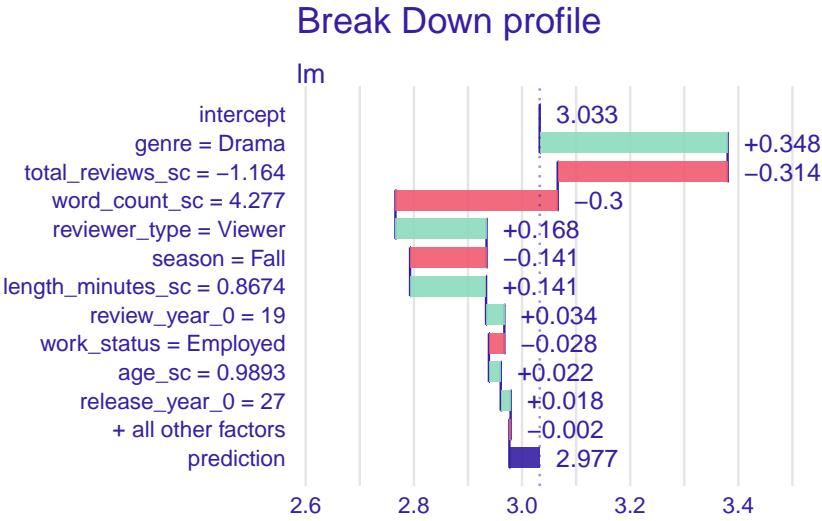


Figure 3.4: R Break Down Plot

### 3.3.1.8 Python

```

break_down_plot = train_explain.predict_parts(
    new_observation = X_features_testing[1],
    type = "break_down"
)

break_down_plot.plot()

```

The Break down plots in Figure 3.4 and `?@fig-py_break-down` show us how each feature contributed to the final prediction for an observation. If a prediction from a model has ever surprised you, this is a great way to see how that prediction actually happened!

### 3.3.1.9 Shap Plots

**Shapley values** are a way to explain the predictions made by machine learning models. They break down a prediction to show the impact of each feature. The Shapley value was originally developed in game theory to determine how much each player in a cooperative game has contributed to the total payoff

of the game. You'll commonly see them used in conjunction with tree-based models, like xgboost, but they can be used with any model.

### 3.3.1.10 R

```
shap_plot = predict_parts(
    train_explain,
    new_observation = testing_data,
    type = "shap")

plot(shap_plot)
```

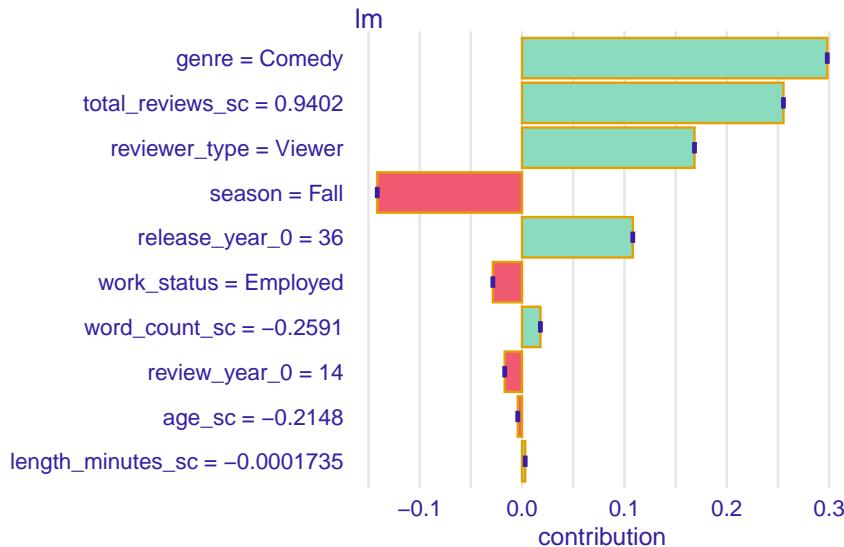


Figure 3.5: R Shap Plot

### 3.3.1.11 Python

```
shap_plot = train_explain.predict_parts(
    new_observation = X_features_testing[1],
    type = "shap"
)

shap_plot.plot()
```

The Shap plots in Figure 3.5 and `?@fig-py_shap` show us how each feature contributed to the final prediction for an observation.

### 3.3.2 Classification

The set-up and functions are exactly the same for classification models, so we want to show you how we can also incorporate information from categorical variables into our explainers.

We'll create our explainer and then look at the **Partial Dependence Plots** for our model, but broken down by `genre`.

#### 3.3.2.1 R

```
train_explain = explain(  
  logistic_model_train,  
  data = training_data[, features],  
  y = training_data$rating_good,  
  verbose = FALSE  
)  
  
train_performance = model_performance(train_explain)  
  
# train_performance # not shown  
  
partial_model_profile = model_profile(train_explain,  
  features,  
  groups = "genre",  
  type = "partial")  
  
plot(partial_model_profile)
```

## Partial Dependence profile

Created for the Im\_Action/Adventure, Im\_Comedy, Im\_Drama, Im\_Horror,

Im_Action/Adventure	Im_Drama	Im_Kids	Im_Romance
Im_Comedy	Im_Horror	Im_Other	Im_Sci-Fi

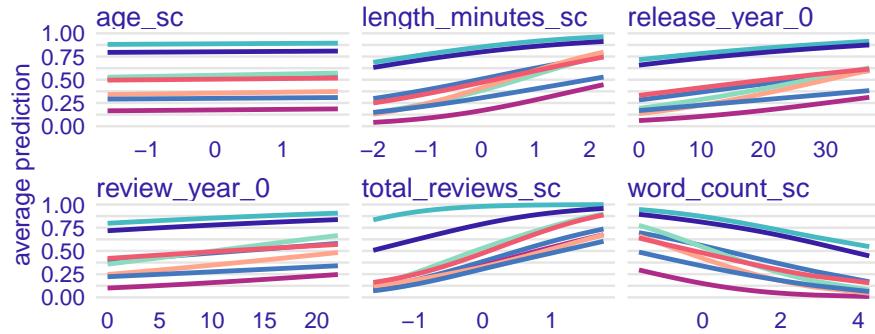


Figure 3.6: R Partial Dependence Plot by Genre

### 3.3.2.2 Python

```

train_explain = dx.Explainer(
    logistic_model_train,
    data = X_features,
    y = training_data.rating_good,
    verbose = False
)

train_performance = train_explain.model_performance()

partial_model_profile = train_explain.model_profile(
    type = "partial"
)

Calculating ceteris paribus:  0% | 0/25 [00:00<?, ?it/s]
Calculating ceteris paribus: 100%|#####| 25/25 [00:00<00:00, 315.19it/s]

partial_model_profile.plot()

```

We can see what R and Python offer us for partial dependence plots in Figure 3.6 and `?@fig-py_partial-plot`. In Figure 3.6, we see those are broken down by the different genres, allowing us to see the differences between genres.

### 3.3.2.3 Variable Importance

We can also look at the variable importance for our classification model. While it operates on the same principle of the regression model, variable importance for classification models is calculated by assessing how a model's AUC changes when a feature is permuted, as opposed to RMSE.

### 3.3.2.4 R

```
model_var_imp = model_parts(train_explain, type = "variable_importance")
plot(model_var_imp)
```

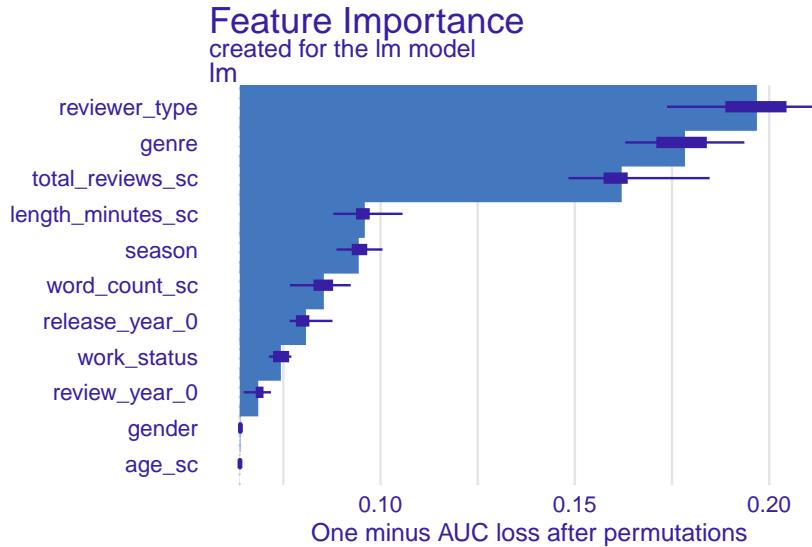


Figure 3.7: R Variable Importance Plot for Classification

### 3.3.2.5 Python

```
model_var_imp = train_explain.model_parts(
    type = "variable_importance"
)
model_var_imp.plot()
```

The variable importance plots in Figure 3.7 and [?@fig-py\\_var-imp-class](#) show us the variables that are the most globally important for making our classifications. How do those variables differ from what we saw in our linear regression model?

Since we have already seen that there isn't much difference between models

with regard to producing these plots, we will leave it up to you to produce localized plots for your classification models!

---

### 3.4 Wrapping Up

It is easy to get caught up in the excitement of creating a model and then using it to make predictions. It is also easy to get caught up in the excitement of seeing a model perform well on a test set. It is much harder to take a step back and ask yourself, “Is this model really doing what I want it to do?” You should always be looking at which variables are pulling the most weight in your model and how predictions are being made.

---

### 3.5 Additional Resources

If this chapter has piqued your curiosity, we would encourage you to check out the following resources.

Even though we did not use the `mlr3` package in this chapter, the **Evaluation and Benchmarking** chapter of the companion book, Applied Machine Learning Using `mlr3` in R<sup>1</sup>, offers a great conceptual take on model metrics and evaluation.

For a more Pythonic look at model evaluation, we would highly recommend going through the sci-kit learn documentation on Model Evaluation<sup>2</sup>. It has you absolutely covered on code examples and concepts.

To get the most out of `DALEX` visualizations, we would recommend checking out Christoph Molnar’s book, Interpretable Machine Learning<sup>3</sup>. It is a great resource for learning more about model explainers and how to use them.

---

<sup>1</sup>[https://mlr3book.mlr-org.com/chapters/chapter3/evaluation\\_and\\_benchmarking.html](https://mlr3book.mlr-org.com/chapters/chapter3/evaluation_and_benchmarking.html)

<sup>2</sup>[https://scikit-learn.org/stable/modules/model\\_evaluation.html](https://scikit-learn.org/stable/modules/model_evaluation.html)

<sup>3</sup><https://christophm.github.io/interpretable-ml-book/>

# 4

---

## *How Did We Get Here?*

---

In our initial linear model, the key **parameters** are the coefficients for each feature. But how do we know what the coefficients are and come to those values? When we run a linear model using some program function, they appear magically, but it's worth knowing a little bit about how they come to be, so let's try and dive a little deeper!

**Model estimation** is the process of finding the parameters associated with a model that allow us to reach a particular modeling goal. Different types of models will have different parameters to estimate, and there are different ways to estimate them. In general though, the goal is the same, find the set of parameters that will lead to the best predictions under the current data modeling context.

With model estimation, we can break things down into the following steps:

1. Start with an **initial guess** for the parameters
2. Calculate the **prediction error**, or some function of it, or some other value that represents our model's **objective**
3. **Update** the guess
4. Repeat steps 2 & 3 until we find a 'best' guess

Pretty straightforward, right? Well, it's not always so simple, but this is the general idea in most applications. In this chapter, we'll show how to do this ourselves to take away the mystery a bit from when you run standard model functions in typical contexts. Hopefully then you'll gain more confidence when you do use them!

### **Estimation vs. Optimization**

We can use **estimation** as general term for finding parameters, while **optimization** can be seen as a term for finding parameters that maximize or minimize some **objective function**, or even a combination of objectives. In some cases we can estimate parameters without optimization, because there is a known way of solving the problem, but in most modeling situations we are going to use some optimization approach to find a 'best' set of parameters.

---

## 4.1 Key Ideas

A few concepts we'll keep using here are fundamental to understanding estimation and optimization. We'll return to these throughout the book, so it's good to get some basic familiarity.

- **Parameters** are the values associated with a model, that we have to estimate.
- **Estimation** is the process of finding the parameters associated with a model.
- The **objective function** produces a value that we want to, for example, maximize or minimize.
- **Prediction error** is the difference between the actual value of the target and the predicted value of the target, and is often used to calculate the objective function.
- **Optimization** is the process of finding the parameters that maximize or minimize some objective function.
- **Model Selection** is the process of choosing the best model from a set of models.

### 4.1.1 Why this matters

When it comes to modeling, even knowing just a little bit about what goes on beyond the scenes is a great demystifier. And if models are less of a mystery, you'll feel more confident in using them. Parts of what you see here are used in almost every common model used for statistics and machine learning, providing you even more of a foundation for understanding what's going on.

### 4.1.2 Good to know

This chapter is more involved than most of the others, and is really for those who like to get their hands dirty. It's all about 'rolling your own', and so we'll be doing a lot of the work ourselves. If you're not one of those types of people that gets much out of that, that's ok, you can skip this chapter and still get a lot out of the rest of the book. But if you're curious about how things work, or you want to be able to do more than just run a function, then we think you'll find the following useful. You'd want to at least have your linear model basics down.

---

## 4.2 Data Setup

For the examples here, we'll use the world happiness dataset for the year 2018. We'll use the happiness score as our target, and we'll use the GDP per capita as our primary feature, though we may throw in some others. Let's take a look at the data here, but for more information see the appendix.

TODO: Link data description

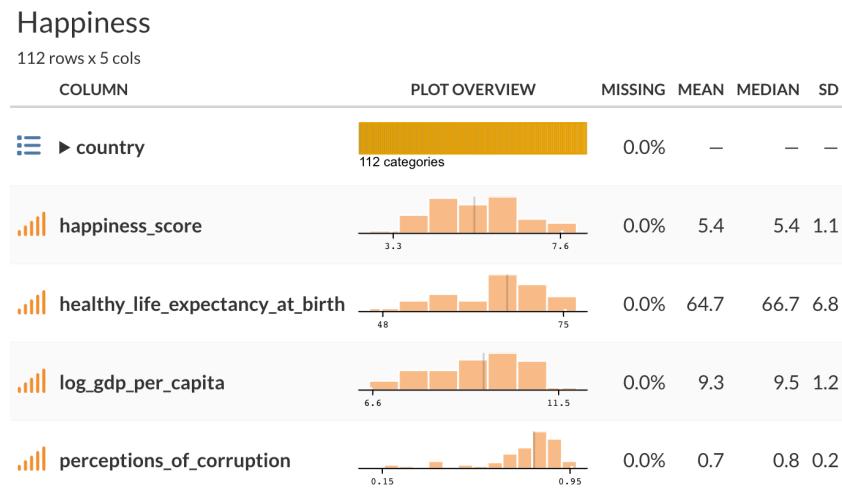


Figure 4.1: World happiness data summary

Our happiness score has values from around 3-7, life expectancy and gdp appear to have some notable variability, and corruption perception is skewed toward lower values. We can also see that the features and target are correlated with each other, which is not surprising.

TODO: check the width of this table in pdf

Table 4.1: Correlation matrix for world happiness data

term	happiness	life_exp	log_gdp_pc	corrupt
happiness	NA	0.78	0.82	-0.47
life_exp	0.78	NA	0.86	-0.34
log_gdp_pc	0.82	0.86	NA	-0.34
corrupt	-0.47	-0.34	-0.34	NA

We'll do some minor cleaning and renaming of columns, and we'll drop any

rows with missing values. We'll also scale the features so that they are on the same scale, which as noted in the data chapter, can help make estimation easier.

#### 4.2.0.1 R

```
df_happiness = read_csv("data/world_happiness_2018.csv") |>
  drop_na() |>
  select(
    country,
    happiness_score,
    healthy_life_expectancy_at_birth,
    log_gdp_per_capita,
    perceptions_of_corruption
  ) |>
  rename(
    happiness = happiness_score,
    life_exp = healthy_life_expectancy_at_birth,
    log_gdp_pc = log_gdp_per_capita,
    corrupt = perceptions_of_corruption
  ) |>
  # put gdp back on original scale before scaling
  mutate(
    gdp_pc = exp(log_gdp_pc),
    across(life_exp:gdp_pc, \((x) scale(x)[,1]) )
  ) |>
  select(-log_gdp_pc) # drop the log version
```

#### 4.2.0.2 Python

```
df_happiness = (
    pd.read_csv('data/world_happiness_2018.csv')
    .dropna()
    .rename(
        columns = {
            'happiness_score': 'happiness',
            'healthy_life_expectancy_at_birth': 'life_exp',
            'log_gdp_per_capita': 'log_gdp_pc',
            'perceptions_of_corruption': 'corrupt'
        }
    )
    .assign(
        gdp_pc = lambda x: np.exp(x['log_gdp_pc']),
    )
    [['country', 'happiness','life_exp', 'gdp_pc', 'corrupt']]
)
```

```
from sklearn.preprocessing import StandardScaler  
  
scaler = StandardScaler()  
  
df_happiness[['life_exp', 'gdp_pc', 'corrupt']] = scaler.fit_transform(  
    df_happiness[['life_exp', 'gdp_pc', 'corrupt']])  
)
```

---

### 4.3 Starting Out by Guessing

So we'll start with a model in which we predict a country's level of happiness by their life expectancy, where if you can expect to live longer, maybe you're probably in a country with better health care, higher incomes, and other important stuff. We'll stick with our simple linear model as well.

As a starting point we can just guess what the parameter should be, but how would we know what to guess? How would we know which guesses are better than others? Let's try a few and see how they do. Let's say that we don't think life expectancy matters, and that most countries are at a happiness value of 4. We can plug this into the model and see what we get:

$$\text{prediction} = 4 + 0 \cdot \text{life\_exp}$$

Alternatively we could use the data to inform our guess. We start with a mean of happiness score, but moving up a standard deviation of life expectancy (roughly  $\sim 1$  years) would move us up a whole point of happiness.

$$\text{prediction} = \overline{\text{happiness}} + 1 \cdot \text{life\_exp}$$

In this case, our offset (or intercept) is the mean of the target, and our coefficient for the scaled life expectancy is 1. This is probably a better guess, since it is at least data driven, but it's still not great. But how do we know it's better?

---

### 4.4 Prediction Error

We can compare the predictions from each guess to the actual values of the target, and observe how far off our predictions are from the observed target.

This difference is the **prediction error**, or in the context of a linear model, they are also called **residuals**. We can express this as:

$$\epsilon = y - \hat{y}$$

error = target – (model based) guess

Not only does this tell us how far off our model prediction is, it gives us a way to compare models. With out measure of prediction error, we can calculate a **metric** for total error for all observations/predictions, or similarly the average error. If one model or parameter set has less total or average error, we can say it's a better model than one that has more. Ideally we'd like to choose a model with the least error, but we'll see that this is not always possible<sup>1</sup>. For now, let's calculate the error for our two guesses. One thing though, if we miss the mark above or below our target, we still want it to count the same in terms of prediction error. In other words, if the true happiness score is 5 and our model predicts 5.5 or 4.5, we want those to count as the same kind of error when we total up our error<sup>2</sup>.

However, if we just take the average, you'll see that it is roughly zero! This is by design for many common models, were we even will explicitly write the formula for the error as coming from a normal distribution with mean of zero. So we need to do something else to get a meaningful error metric. One way we can get around this is to use the squared error value, or maybe the absolute value. We'll use squared error here, and we'll calculate the mean of the squared errors for all our predictions. We'll do this for our two models above.

#### 4.4.0.1 R

```
y = df_happiness$happiness

# Calculate the error for the guess of 4
prediction = 4
mse_four = mean((y - prediction)^2)

# Calculate the error for our other guess
prediction = mean(y) + 1 * df_happiness$life_exp
mse_other = mean((y - prediction)^2)
```

---

<sup>1</sup>It turns out that our error metric is itself an *estimate* of the true error. We'll get more into this later, but for now this means that we can't ever know the true error, and so we can't ever really know the best or true model. However, we can still choose a good or better model relative to others based on our estimate.

<sup>2</sup>We don't have to do it this way, but it's the default in most scenarios. As an example, maybe for your situation overshooting is worse than undershooting, and so you might want to use an approach that would weight those errors more heavily.

#### 4.4.0.2 Python

```
y = df_happiness['happiness']

# Calculate the error for the guess of four
prediction = 4
mse_four = np.mean((y - prediction)**2)

# Calculate the error for our other guess
prediction = y.mean() + 1 * df_happiness['life_exp']
mse_other = np.mean((y - prediction)**2)
```

Now let's look at our **Mean Squared Error** (MSE), and we'll also inspect the square root of it, or the **Root Mean Squared Error**, as that puts things back on the original target scale, and tells us the standard deviation of our prediction errors. We also add the **Mean Absolute Error** (MAE) as another metric with straightforward interpretation. Inspecting the metrics, we can see that we are off on average by over a point for our '#4' model, but notably less when guessing the mean.

Table 4.2: Comparison of error metrics for two models

Model	MSE	RMSE	MAE	RMSE % drop	MAE % drop
#4	3.36	1.83	1.52		
Other	0.50	0.71	0.58	61%	62%

We can see that the 'other' model is not only better, but results in a 61% drop in RMSE, and similar for MAE. We'd definitely prefer that model over the '#4' model. Furthermore, we can see how we can compare models in a general fashion.

Now all of this is useful, and at least we can say one model is better than another. But you're probably hoping there is an easier way to do get a good guess for our model parameters, especially when we have possibly dozens of features and/or parameters to keep track of, and there is!

---

## 4.5 Ordinary Least Squares

For a simple linear model, we can estimate the parameters in several ways, but the most common is to use the **Ordinary Least Squares (OLS)** method. OLS is a method of estimating the coefficients that minimizes the sum of the

squared errors, which we've just been doing in the previous section<sup>3</sup>. In other words, it finds the coefficients that minimize the sum of the squared differences between the predicted values and the actual values. We can express this as:

$$\text{Value} = \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (4.1)$$

Where  $y_i$  is the actual value of the target for observation  $i$ , and  $\hat{y}_i$  is the predicted value from the model. The sum of the squared errors is also called the **residual sum of squares** (RSS), as opposed to the total sums of squares (i.e. the variance of the target), and the part explained by the model (model or explained sums of squares). The OLS method finds the coefficients that minimize the sum of the squared differences between the predicted values and the actual values. It's called *ordinary* least squares because there are other least squares methods - generalized least squares, weighted least squares, and others, but we don't need to worry about that for now. What matters is that we have a way to estimate the coefficients that minimizes the sum of the squared errors.

The resulting value - the sum or mean of the squared errors - can be referred to as our **objective value**, while the **objective function** is just the process of taking the predictions and observed target values as inputs and totaling up their squared differences to be returned as an output. We can use this value to find the best parameters for a specific model, as well as compare models with different parameters, such as a model with additional features versus one with fewer. We can also use this value to compare different types of models that are using the same objective function, such as a linear model and a decision tree model.

Let's calculate the OLS estimate for our model. From our steps above, we need guesses and a way to update them. For now, we can just provide a bunch of guesses, and just move along from one set to the next, and ultimately just choose whichever has the lowest value.

#### 4.5.0.1 R

```
# for later comparison
model_happy = lm(happiness ~ life_exp, data = df_happiness)

ols = function(X, y, par, sum_sq = FALSE) {
  # add a column of 1s for the intercept
  X = cbind(1, X)
```

---

<sup>3</sup>Some disciplines seem confuse models with estimation methods and link functions. It doesn't really make sense, nor is informative, to call something an OLS model or a logit model. Many models are estimated using a least squares approach, and different types of models use a logit link.

```

# Calculate the predicted values
y_hat = X %*% par # %*% is matrix multiplication

# Calculate the error
error = y - y_hat

# Calculate the value as sum or mean squared error
value = crossprod(error) # crossprod is matrix multiplication

if (!sum_sq) {
  value = value / nrow(X)
}

# Return the value
return(value)
}

# create a grid of guesses
guesses = crossing(
  b0 = seq(1, 7, 0.1),
  b1 = seq(-1, 1, 0.1)
)

# Example for one guess
ols(
  X = df_happiness$life_exp,
  y = df_happiness$happiness,
  par = unlist(guesses[1, ])
)
[,1]
[1,] 23.777

```

#### 4.5.0.2 Python

```

# for later comparison
model_happy = smf.ols('happiness ~ life_exp', data = df_happiness)
model_happy = model_happy.fit()

def ols(par, X, y, sum = False):
  # add a column of 1s for the intercept
  X = np.c_[np.ones(X.shape[0]), X]

  # Calculate the predicted values
  y_hat = X @ par

```

```

# Calculate the error
value = np.sum((y - y_hat)**2)

# Calculate the value as sum or average
if not sum:
    value = value / X.shape[0]

# Return the value
return(value)

# create a grid of guesses
from itertools import product

guesses = pd.DataFrame(
    product(
        np.arange(1, 7, 0.1),
        np.arange(-1, 1, 0.1)
    ),
    columns = ['b0', 'b1']
)

# Example for one guess
ols(
    par = guesses.iloc[0,:],
    X = df_happiness['life_exp'],
    y = df_happiness['happiness']
)

```

23.793842044979073

Now we want to calculate the loss for each guess and find which one gives us the minimum function value. Note that above, we could get the total or mean squared error by setting the `sum` parameter to `TRUE` or `FALSE`. Either is fine, but it's more common to use the mean, which is a little more understandable - how far do our guess deviate from the true value on average? In the following darker suggests a better mean squared error result from our approach.

If we inspect our results from the built-in functions, we had estimates of 5.44 and 0.89 for our coefficients versus the best guess from our approach of 5.4 and 0.9. These are very similar but not exactly the same, but this is mostly due to the granularity of our guesses. Even so, in the end we can see that we get pretty dang close to what our basic `lm` or `statsmodels` functions would get us. Pretty neat!

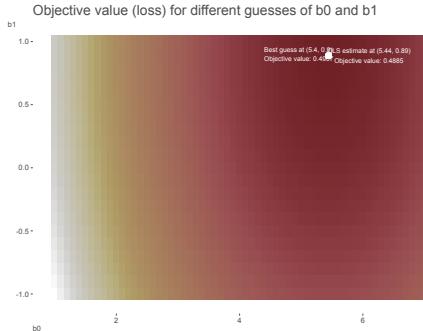


Figure 4.2: Results of parameter search

### Estimation as ‘Learning’

Estimation, and/or optimization, can be seen as the process of a model learning which parameters will best allow the predictions to match the observed data, and hopefully, predict as-yet-unseen future data. This is a very common way to think about estimation in machine learning, and it is a useful way to think about our simple linear model also.

One thing to keep in mind is that it is not a magical process. It takes good data, a good idea (model), and an appropriate estimation method to get good results.

---

## 4.6 Optimization

Before we get into other objective functions, let’s think about a better way to find the best parameters for our model. Rather than just guessing, we can use a more systematic approach, and thankfully, there are tools out there to help us. We just use a function like our OLS function, give it a starting point, and let the algorithms do the rest! Thanks to some nifty approaches to making better guesses, these tools eventually arrive at a pretty good set of parameters. Well, they usually do, but not always- nothing’s perfect! But they are pretty good, and they are a lot better than guessing. Let’s see how we can use one of these tools to find the best parameters for our model.

Previously we created a set of guesses to search over to see which set of parameters resulted in prediction that matched the data best. What we did is called a **grid search**, and it is a bit of a brute force approach to finding the best fitting model. You can imagine a couple of unfortunate or problematic scenarios, such as having a very large number of parameters, or that our

specified range doesn't allow us to get to the right sets of parameters, or we specify a very large range, but the best fitting model is within a very narrow part of that range. In any of these cases we waste a lot of time, and may not find an optimal solution.

In general, we can think of **optimization** as a way to find the best parameters for our model. We start with an initial guess, see how well it does in terms of our objective function, and then try to improve it with a new guess. We continue to do so until a stopping point is reached. Here is an example.

- **Start with an initial guess** for the parameters
- Calculate the objective function given the parameters
- **Update the parameters** to a new guess (that hopefully improves the objective function)
- Calculate the objective function given the new parameters
- **Repeat** until the improvement is small enough, or we reach the desired number of iterations we want to attempt

This is what we described before with estimation in general. The key idea now is how we *update* the old parameters with a new guess at each iteration. Different **optimization algorithms** use different approaches to find the updated parameters. At some point, either the improvement is no longer practical, often referred to as our **tolerance**, or we reach a maximum number of iterations we want to attempt, and either of these is something we can set ourselves. If we meet the terms of our objective, we say that our model has **converged**. Sometimes, the number of iterations is not enough for us to reach convergence in terms of tolerance, and we have to try again with a different set of parameters, a different algorithm, maybe use some data transformations, or something else.

So let's try it out! Both R and Python offer a function where we can specify the objective function, and it will try to find the best parameters for us. It needs several inputs:

- the objective function
- the initial guess for the parameters to get things going
- inputs to the objective function
- options for the optimization process, e.g. algorithm, maximum number of iterations, etc.

With these inputs, we'll let the optimization functions do the rest of the work. We'll also compare our results to the built-in functions to make sure we're on the right track.

#### 4.6.0.1 R

We'll use the `optim` function in R.

```
our_result = optim(
```

```

par      = c(1, 0),
fn       = ols,
X        = df_happiness$life_exp,
y        = df_happiness$happiness,
method   = "BFGS", # optimization algorithm
control  = list(  # specify tolerance, max iter, etc. here
    tol   = 1e-6,
    maxit = 500
)
)

# our_result

```

#### 4.6.0.2 Python

We'll use the `minimize` function in Python.

```

from scipy.optimize import minimize

our_result = minimize(
    fun      = ols,
    x0      = np.array([1., 0.]),
    args    = (
        np.array(df_happiness['life_exp']),
        np.array(df_happiness['happiness'])
    ),
    method  = 'BFGS', # optimization algorithm
    tol     = 1e-6,   # tolerance
    options = {
        'maxiter': 500
    }
)

# our_result

```

Table 4.3: Comparison of our results to built-in function

Parameter	Built-in	Our Result
Intercept	5.4450	5.4450
Life Exp. Coef.	0.8880	0.8880
Objective/MSE	0.4890	0.4890

So our little function and the right tool allows us to come up with the same thing as base R and `statsmodels`! I hope you're feeling pretty good at this point because you should! You just proved you could do what seemed before to be

like magic, but really all it took is just a little knowledge about some key concepts. Let's try some more!

### A Note on Terminology

The objective function is often called the **loss function**, and sometimes the **cost function**. However, these both imply that we are trying to minimize the function, which is not always the case<sup>4</sup>, and it's arbitrary whether you want to minimize or maximize the function. In fact, some people will minimize the *negative* likelihood when using *maximum* likelihood! As such we'll try to stick to the more neutral term 'objective function', but you may see the other terms used interchangeably in this text. In addition, some packages will use the term **metric** to refer a value that you might want to examine as well, or even use to compare models. For example, the MSE is a metric, but it may also be the objective function we are trying to minimize. Other metrics we could calculate without being the objective might be Adjusted R-squared and mean absolute error. We could also use MSE as the objective, but use percentage drop in error from baseline when selecting among several models that minimized MSE. This can be very confusing when starting out! We'll try to stick to the term *metric* for additional values that we might want to examine separate from the *objective function value*.

## 4.7 Maximum Likelihood

In our example thus far, we have been minimizing the specific objective (or loss) function, which basically takes our parameter estimates, produces a prediction, and returns the sum or mean of the squared errors. But this is just one approach we could take. Now we'd like you to think about the **data generating process**. We have a model that says happiness is a function of life expectancy, but more specifically, let's think about how the observed value of the happiness score is generated in a statistical sense. In particular, what kind of probability distribution might be involved? Ignoring the model, we might think that each happiness value is generated by some random process, and that the process is the same for each observation. Let's assume that random process is a normal distribution. So something like this would describe it mathematically:

<sup>4</sup>You may find that some packages will only minimize (or maximize) a function, possibly one that you come up with yourself. It'd be nice if they did this internally, or allowed the user to specify the direction like most packages, but you'll need to take care when implementing your own metrics.

$$\text{happiness} \sim N(\mu, \sigma)$$

where  $\mu$  is the mean of the happiness and  $\sigma$  is the standard deviation, or in other words, we can think of happiness as a random variable that is drawn from a normal distribution with  $\mu$  and  $\sigma$  as the parameters of that distribution.

Let's apply this idea to our linear model setting. In this case, the mean is a function of life expectancy, and we're not sure what the standard deviation is, but we can go ahead and write our model as follows.

$$\begin{aligned}\mu &= \beta_0 + \beta_1 * \text{life\_exp} \\ \text{happiness} &\sim N(\mu, \sigma)\end{aligned}$$

Now, we can think of the model as a way to estimate the parameters of the normal distribution, but we have an additional parameter to estimate. We still have our previous coefficients, but now we need to estimate  $\sigma$ , which is basically our RMSE, as well. But we still have to think of things a little differently. When we compare our prediction to the observed value, we don't look at the simple difference, but we are still interested in the discrepancy between the two. So now we think about the **likelihood** of observing the happiness score given our prediction, which is based on the estimated parameters, i.e. given the  $\mu$  and  $\sigma$ , and  $\mu$  is a function of the coefficients and life expectancy. We can write this as:

$$\Pr(\text{happiness} | \text{life\_exp}, \beta_0, \beta_1, \sigma)$$

$$\Pr(\text{happiness} | \mu, \sigma)$$

Even more generally, the likelihood gives us a sense of the probability of the observed data given the parameter estimates  $\theta$ .

$$\Pr(\text{Data} | \theta)$$

Here is a simple code demo to get a likelihood in the context of our model. The values you see are referred to statistically as probability density values, and they are technically not probabilities, but rather the probability density, or **relative likelihood**, at that observation<sup>5</sup>. For your conceptual understanding, if it makes it easier, you can think of them in the same was as you do probabilities, but just know that technically they are not.

TODO: UPDATE VALUES WHEN DEMO IS SETTLED

---

<sup>5</sup>The actual probability of a *specific value* is 0, but the probability of a range of values is not 0. You can find out more about likelihoods and probabilities at the discussion here<sup>6</sup>, but in general many traditional statistical texts will cover this also.

#### 4.7.0.1 R

```
# two example life expectancy scores, mean and 1 sd above
life_expectancy = c(0, 1)

# observed happiness scores
happiness = c(4, 5.2)

# predicted happiness with rounded coeffs
mu = 5 + 1 * life_expectancy

# just a guess for sigma
sigma = .5

# likelihood for each observation
L = dnorm(happiness, mean = mu, sd = sigma)
L

[1] 0.1079819 0.2218417
```

#### 4.7.0.2 Python

```
from scipy.stats import norm

# two example life expectancy scores, mean and 1 sd above
life_expectancy = np.array([0, 1])

# observed happiness scores
happiness = np.array([4, 5.2])

# predicted happiness with rounded coeffs
mu = 5 + 1 * life_expectancy

# just a guess for sigma
sigma = .5

# likelihood for each observation
L = norm.pdf(happiness, loc = mu, scale = sigma)
L

array([0.1080, 0.2218])
```

So, given a guess at the parameters, and an assumption about the distribution of the data, we can calculate the likelihood of observing each data point, and sum those up, just like we did with our squared errors. In theory, we'd deal with the product of each likelihood, but in practice we sum the log of the likelihood, otherwise values would get too small for our computers to handle.

Here is a corresponding function we can use to calculate the likelihood of the data given our parameters. Note that the actual likelihood value returned isn't really interpretable, just that higher is better from a maximization standpoint. But we can use it to compare models with different sets of parameter guesses. Even if our total likelihoods under comparison are negative, we prefer the model with the relatively higher likelihood. As we just demonstrated, we'll use `optim` to help us get good guesses<sup>7</sup>.

#### 4.7.0.3 R

```
likelihood = function(par, X, y) {
  X = cbind(1, X)
  # setup
  beta = par[-1] # coefficients
  sigma = exp(par[1]) # error sd, exp keeps positive

  N = nrow(X)

  LP = X %*% beta # linear predictor
  mu = LP # identity link in the glm sense

  # calculate (log) likelihood
  ll = dnorm(y, mean = mu, sd = sigma, log = TRUE)
  -sum(ll) # for minimization
}

our_result = optim(
  par = c(1, 0, 0),
  fn = likelihood,
  X = df_happiness$life_exp,
  y = df_happiness$happiness
)

# our_result
```

#### 4.7.0.4 Python

```
def likelihood(par, X, y):
    # add a column of 1s for the intercept
```

---

<sup>7</sup>Those who have experience here will notice we aren't putting a lower bound on `sigma`. You typically want to do this otherwise you may get nonsensical results. You can do this by using the `lower` parameter in `optim` with an algorithm that uses boundaries, or even more simply by exponentiating the parameter, i.e. `exp(par[1])`. Just remember that the returned value will be on the log scale, so you'll have to exponentiate it to get to the correct scale. We leave this detail out of the code for now to keep things simple.

```

X = np.c_[np.ones(X.shape[0]), X]

# setup
beta  = par[1:]          # coefficients
sigma = np.exp(par[0])   # error sd, exp keeps positive

N = X.shape[0]

LP = X @ beta            # linear predictor
mu = LP                  # identity link in the glm sense

# calculate (log) likelihood
ll = norm.logpdf(y, loc = mu, scale = sigma)
return(-np.sum(ll))

our_result = minimize(
    fun = likelihood,
    x0  = np.array([1, 0, 0]),
    args = (
        np.array(df_happiness['life_exp']),
        np.array(df_happiness['happiness'])
    )
)
)

```

How would we switch to a maximum likelihood approach using readily available functions? In both R and Python you can switch to using `glm` and `GLM` respectively as a start. We can use different likelihoods corresponding to the binomial, poisson and other distributions. Still other packages would allow even more distributions for consideration. In general, we choose a distribution that we feel best reflects the data generating process. For binary targets for example, we typically would feel a bernoulli or binomial distribution is appropriate. For count data, we might choose a poisson or negative binomial distribution. For targets that fall between 0 and 1, we might go for a beta distribution. There are many distributions, and even when some might feel more appropriate, we might choose another for convenience. Some distributions tend toward a normal (a.k.a. gaussian) distribution depending on various factors, while others are special cases of more general distributions. For example, the exponential distribution is a special case of the gamma distribution, and a cauchy is equivalent to a t distribution with 1 degree of freedom, and the t tends toward a normal with increasing degrees of freedom. Here is a visualization of the relationships among some of the more common distributions Wikipedia (2023).

TODO: fig needs work for pdf

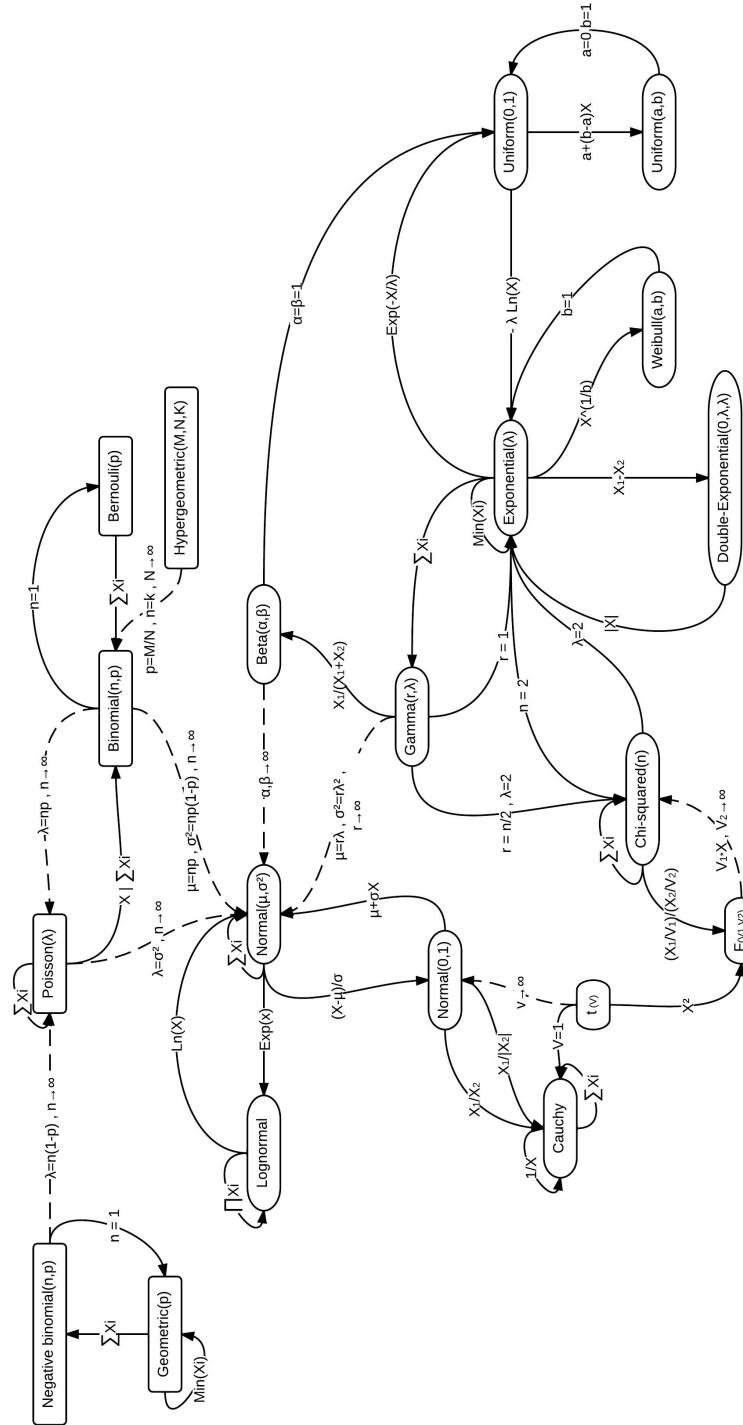


Figure 4.3: Relationships among some of probability distributions

Here are examples of standard GLM functions in R and Python

#### 4.7.0.5 R

```
glm(happiness ~ life_exp, data = df_happiness, family = gaussian)
glm(binary_target ~ x1 + x2, data = some_data, family = binomial)
glm(count ~ x1 + x2, data = some_data, family = poisson)
```

#### 4.7.0.6 Python

```
import statsmodels.formula.api as smf

smf.glm(
    'happiness ~ life_exp',
    data = df_happiness,
    family = sm.families.Gaussian()
)

smf.glm(
    'binary_target ~ x1 + x2',
    data = some_data,
    family = sm.families.Binomial()
)

smf.glm(
    'count ~ x1 + x2',
    data = some_data,
    family = sm.families.Poisson()
)
```

With that in mind, we can compare our result to a built-in function that has capabilities beyond OLS. As before, we're duplicating the basic `glm` result. We show more decimal places on the log likelihood estimate to prove we aren't getting *exactly* the same result

Table 4.4: Comparison of our results to built-in function

Parameter	Built-in	Our Result
Intercept	5.44	5.44
Life Exp. Coef.	0.89	0.89
Sigma	0.71	0.70 <sup>1</sup>
LogLik (neg)	118.80	118.80

<sup>1</sup>Parameter estimate is exponentiated

Let's think more about what's going on here. It turns out that our objective

function defines a space or surface. We can think of it as a landscape, and we are trying to find the lowest point on that landscape. We can then think of our guesses as points on that landscape, and we are trying to find the lowest point. Let's start get a sense of this with the following visualization, based on a single parameter. The data is drawn from Poisson distributed variable with true mean  $\theta = 5$ . We note the calculated likelihood increases as we estimate values for  $\theta$  closer to 5, or more precisely, whatever the mean observed value is for the data. However, with more and more data, the final ML estimate will converge on the true value. Model estimation finds that maximum on the curve, and optimization algorithms are the means to find it.

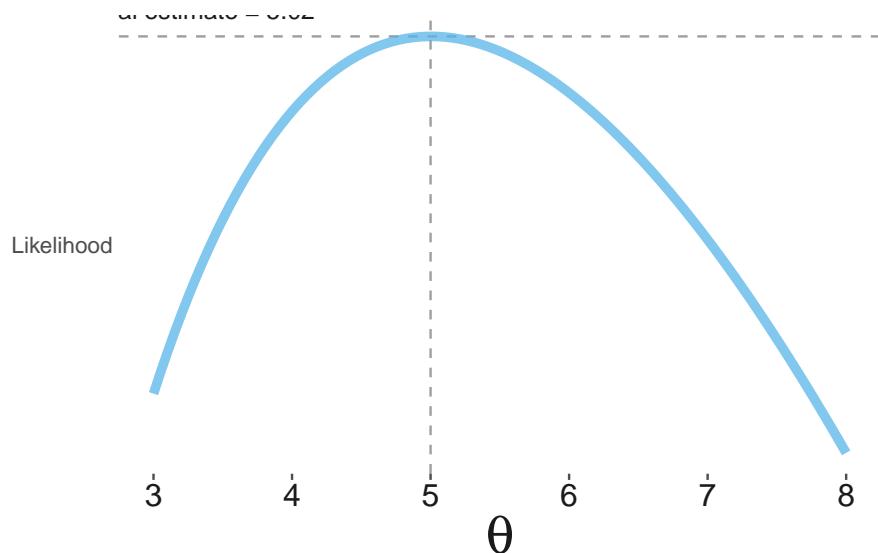


Figure 4.4: Likelihood function one parameter

Now let's add a parameter. If we have more than one parameter, we now have a surface to deal with. Given some starting point, an optimization procedure then travels along the surface looking for a minimum/maximum point. For simpler settings such as this, we can visualize the likelihood surface and its minimum point. However, even our simple demo model has three parameters plus the likelihood, so would be difficult to visualize without additional complexity. To get around this, we show the results for an alternate model where happiness is standardized also, which means the intercept is zero<sup>8</sup>, and we don't have to show that.

---

<sup>8</sup>Linear regression will settle on a line that cuts through the means, and when standardizing the mean of the features and target are both zero, so the line goes through the origin.

TODO: CANT DO INTERACTIVE WITH PDF/LATEX. NEED WORKAROUND.

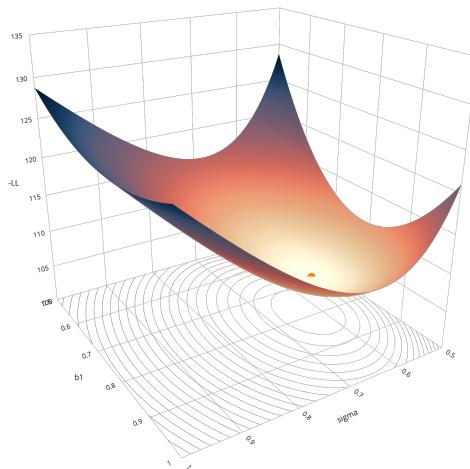


Figure 4.5: Likelihood surface for two parameters

We can also see the path our estimates take, starting at a rather poor point, but quickly updating to estimates that result in a better likelihood value. We also see little exploratory jumps creating a star like pattern, before things ultimately settle to the best values. In general, these updates and paths are dependent on the optimization algorithm one uses.

It turns out that in the case of a normal distribution, the maximum likelihood estimate of the standard deviation is the estimate as the standard deviation of the residuals. Furthermore, the maximum likelihood estimates and OLS estimates converge to the same estimates as the sample size increases. For any data of significance, these estimates are indistinguishable, and the OLS estimate is the maximum likelihood estimate for linear regression.

#### 4.7.0.7 Additional Thoughts on Maximum Likelihood

TODO: Remove?

One of the key things to note is that maximum likelihood is an estimation technique that relies on specifying the probability distribution that serves as the data generating process. Maximum likelihood allows us to be explicit

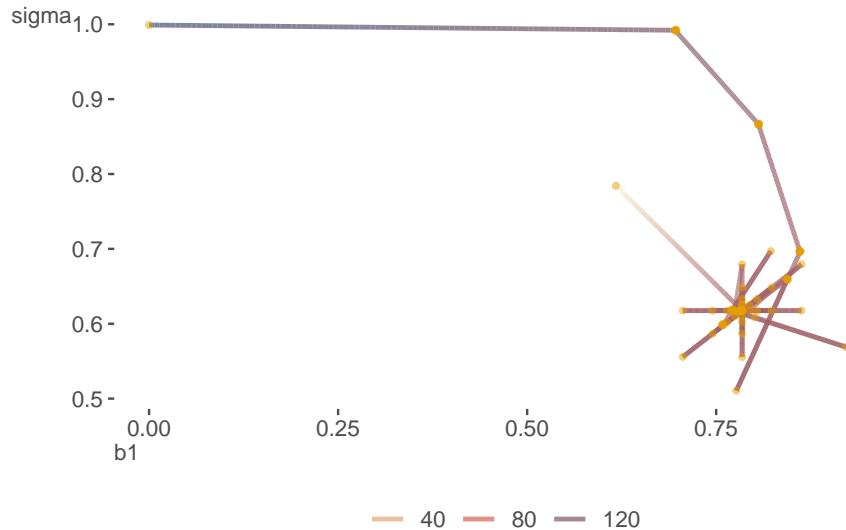


Figure 4.6: Optimization path for two parameters

about why we think those target values are the way they are. The likelihood also serves as a fundamental part of Bayesian analysis, which we'll discuss more later. In general, maximum likelihood is a powerful technique that can be used in many contexts, and likelihoods can be used as the objective for many machine learning algorithms as well.

## 4.8 Estimation: Quick Review

TODO: MOVE WHERE? NEEDED?

At this point we understand a few things:

- Parameters are the values associated with a model
- Objective functions specify a modeling goal with which to estimate the parameters.
- Estimation is a way of finding the best model, i.e. parameters that help us achieve a goal.
- Optimization is the process of finding the parameters that maximize or minimize some objective function
- The likelihood is an alternate way to assess the match of data and model, and allows us to compare the relative fits of models

## 4.9 Penalized Objectives

TODO: MOVE TO AFTER CLASSIFICATION?

One thing we may want to take into account of with our models is their complexity, especially in the context of **overfitting**. We talk about this with machine learning also, but the basic idea is that we can get too close to the data we have, such that when we try to predict on new data, our performance suffers or even gets worse than a simpler model. In other words, we are not generalizing well. One way to deal with this is to penalize the objective function value for complexity, or at least favor simpler models that might do as well. In some contexts this is called **regularization**, and in other contexts **shrinkage**, since the parameter estimates are typically shrunk toward some specific value (e.g., zero).

As a starting point, in our basic linear model we can add a penalty that is applied to the size of coefficients. This is called **ridge regression**, or, more mathily, as **L2 regularization**. The penalty is just the sum of the squared coefficients multiplied by a some value, which we call  $\lambda$ . We can write this formally as:

$$\text{Value} = \sum_{i=1}^n (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^p \beta_j^2 \quad (4.2)$$

The first part is the same as before (Equation 4.1), but the second part is the penalty for  $p$  features. The penalty is the sum of the squared coefficients multiplied by some value, which we call  $\lambda$ . This is an additional model parameter that we typically want to estimate in some fashion, e.g. through cross-validation. This kind of parameter is often called a hyperparameter, mostly just to distinguish it from those that may be of actual interest. For example, we could probably care less what the actual value for  $\lambda$  is, but we would still be interested in the coefficients.

Interestingly, as you'll notice that this is just OLS+, you might be wondering how our results or interpretation might change. Well for starters, L2 regularization is not limited to linear regression, so just keep that in mind. But also, if we know that OLS produces **unbiased** estimates if assumptions of linear regression are met, that means, if these aren't the same estimates, they must be biased, right? Your are correct! As we talk about with machine learning (Section 7.4), the bias-variance tradeoff is a key concept in machine learning, and this is a good example of that. We are introducing some bias in order to reduce the variance. In other words, we are willing to accept some bias in order to get a model that generalizes better.

Another common penalty that is the sum of the absolute value of the coefficients, which is called **lasso regression** or **L1 regularization**. An interesting property of the lasso is that in typical implementations, it will potentially zero out coefficients, which is the same as dropping the feature from the model altogether. This is a form of **feature selection** or **variable selection**. The true values are never zero, but if we want to use a ‘best subset’ of features, this is one way we could do so. We can write the lasso objective as:

$$\text{Value} = \sum_{i=1}^n (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^p |\beta_j| \quad (4.3)$$

But let’s get to a code example to make sure we understand this better! Here is an example of a function that calculates the ridge objective. To make things interesting, let’s add the other features we talked about regarding GDP per capita and perceptions of corruption.

#### 4.9.0.1 R

```
ridge = function(par, X, y, lambda = 0) {
  # add a column of 1s for the intercept
  X = cbind(1, X)

  # Calculate the predicted values
  mu = X %*% par # %*% is matrix multiplication

  # Calculate the value as sum squared error
  error = crossprod(y - mu)

  # Add the penalty
  value = error + lambda * crossprod(par)

  return(value)
}

our_result = optim(
  par = c(0, 0, 0, 0),
  fn = ridge,
  X = df_happiness |> select(-happiness, -country) |> as.matrix(),
  y = df_happiness$happiness,
  lambda = 0.1,
  method = "BFGS"
)
```

#### 4.9.0.2 Python

```
# we use lambda_ because lambda is a reserved word in python
def ridge(par, X, y, lambda_ = 0):
    # add a column of 1s for the intercept
    X = np.c_[np.ones(X.shape[0]), X]

    # Calculate the predicted values
    mu = X @ par

    # Calculate the error
    value = np.sum((y - mu)**2)

    # Add the penalty
    value = value + lambda_ * np.sum(par**2)

    return(value)

our_result = minimize(
    fun = ridge,
    x0 = np.array([0, 0, 0, 0]),
    args = (
        np.array(df_happiness.drop(columns=['happiness', 'country'])),
        np.array(df_happiness['happiness']),
        0.1
    )
)
```

We can compare this to built-in functions as we have before, and can see that the results are very similar, but not exactly the same. We would not worry about such differences in practice, but the main point is again, we can use simple functions that do just about as well as any what we'd get from package output.

Table 4.5: Comparison of ridge regression results

Parameter	Built-in <sup>1</sup>	Our Result
Intercept	5.44	5.44
Life Exp. Coef.	0.49	0.52
Corrupt	-0.12	-0.11
GDP_PC	0.42	0.44

<sup>1</sup>Showing results from R glmnet package with alpha = 0, lambda = .1

💡 It turns out that, given a set  $\lambda$  penalty, ridge regression estimates need not be estimated, as there is an analytical result. See a demo<sup>a</sup> for more.

<sup>a</sup><https://m-clark.github.io/models-by-example/penalized-maximum-likelihood.htm>

## 4.10 Classification

So far we've been assuming a continuous target, but what if we have a categorical target? Now we have to learn a bunch of new stuff for that situation, right? Actually, no! *When we want to model categorical targets, conceptually nothing changes* - we can still have an objective function that maximizes or minimizes some goal, use the same algorithms to estimate parameters, etc. However, we need to think about how we can do this in a way that makes sense for the target.

### 4.10.1 Misclassification

A straightforward correspondence to MSE is a function that minimizes classification error (or maximizes accuracy). In other words, we can think of the objective function as the proportion of incorrect classifications. This is called the **misclassification rate**. We can write this as:

$$\text{Loss} = \frac{1}{n} \sum_{i=1}^n \mathbb{1}(y_i \neq \hat{y}_i)$$

Where  $y_i$  is the actual value of the target for observation  $i$ , arbitrarily coded as 1 or 0, and  $\hat{y}_i$  is the predicted class from the model. The  $\mathbb{1}$  is an indicator function that returns 1 if the condition is true, and 0 otherwise. In other words, we are counting the number of times the predicted value is not equal to the actual value, and dividing by the number of observations. Very straightforward, so let's do this ourselves!

#### 4.10.1.1 R

```
# misclassification rate
misclassification = function(par, X, y, class_threshold = .5) {
  X = cbind(1, X)
  # Calculate the predicted values
  mu = X %*% par # %*% is matrix multiplication
```

```

# Convert to a probability ('sigmoid' function)
p = 1 / (1 + exp(-mu))

# Convert to a class
predicted_class = as.integer(
  ifelse(p > class_threshold, "good", "bad")
)

# Calculate the error
error = y - predicted_class

return(mean(error))
}

```

#### 4.10.1.2 Python

```

def misclassification_rate(par, X, y, class_threshold = .5):
    # add a column of 1s for the intercept
    X = np.c_[np.ones(X.shape[0]), X]

    # Calculate the predicted values
    mu = X @ par

    # Convert to a probability ('sigmoid' function)
    p = 1 / (1 + np.exp(-mu))

    # Convert to a class
    predicted_class = np.where(p > class_threshold, 1, 0)

    # Calculate the error
    error = y - predicted_class

    return(np.mean(error))

```

We'll leave it as an exercise to the reader to play around with this, as the next objective function is more commonly used. But at least you can see how easy it can be to switch to the classification case.

#### 4.10.2 Log loss

Another approach is to use the **log loss**, sometimes called logistic loss or cross-entropy. If we have just the binary case it is:

$$\text{Loss} = - \sum_{i=1}^n y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)$$

Where  $y_i$  is the actual value of the target for observation  $i$ , and  $\hat{y}_i$  is the predicted value from the model (essentially a probability). It turns out that *this is the same as log-likelihood used in a maximum likelihood approach for logistic regression*, made negative for minimization. We typically prefer this objective function to classification error because it results in a *smooth* optimization surface, like in the visualization we showed before for maximum likelihood (Figure 4.5), which means it is *differentiable* in mathematical sense. This is important because it allows us to use optimization algorithms that rely on derivatives in updating the parameter estimates. You don't really need to get into that too much, but just know that it is a good thing. Here's some code to try out.

#### 4.10.2.1 R

```
objective = function(par, X, y) {
  X = cbind(1, X)

  # Calculate the predicted values on the raw scale
  y_hat = X %*% par

  # Convert to a probability ('sigmoid' function)
  y_hat = 1 / (1 + exp(-y_hat))

  # likelihood (or dbinom(y, size = 1, prob = y_hat, log = TRUE))
  ll = y * log(y_hat) + (1 - y) * log(1 - y_hat)

  return(sum(-ll))
}
```

#### 4.10.2.2 Python

```
def objective(par, X, y):
  # add a column of 1s for the intercept
  X = np.c_[np.ones(X.shape[0]), X]

  # Calculate the predicted values
  y_hat = X @ par

  # Convert to a probability ('sigmoid' function)
  y_hat = 1 / (1 + np.exp(-y_hat))

  # likelihood
```

```

ll = y * np.log(y_hat) + (1 - y) * np.log(1 - y_hat)

return(-np.sum(ll))

```

Let's go ahead and demonstrate this. Let's go back to our movie review data, but we'll use a version of our rating where a movie is 'good' if the rating is 3 or greater, and 'bad' otherwise, which we have in our processed version of the data. Our features will be the review year (starting at zero), reviewer age, and word count. Let's use our previous optimization functions, and compare our results to the built-in complements.

#### 4.10.2.3 R

```

df_reviews_pr = read_csv("data/movie_reviews_processed.csv")

mod_logloss = optim(
  par = c(0, 0, 0, 0),
  fn = objective,
  X = df_reviews_pr |>
    select(review_year_0, age_sc, word_count_sc) |>
    as.matrix(),
  y = df_reviews_pr$rating_good
)

mod_glm = glm(
  rating_good ~ review_year_0 + age_sc + word_count_sc,
  data   = df_reviews_pr,
  family = binomial
)

```

#### 4.10.2.4 Python

```

from scipy.optimize import minimize

mod_logloss = minimize(
  objective,
  x0 = np.array([0, 0, 0, 0]),
  args = (
    df_reviews_pr[['review_year_0', 'age_sc', 'word_count_sc']],
    df_reviews_pr['rating_good']
  )
)

mod_glm = smf.glm(
  'rating_good ~ review_year_0 + age_sc + word_count_sc',
  data   = df_reviews_pr,

```

```
family = sm.families.Binomial()
).fit(method = 'lbfgs')
```

Once again we can see that the results are very similar, but not exactly the same, though actually have to go out several decimal places before we start seeing differences between our result and the built-in function.

Table 4.6: Comparison of log loss results

name	Ours	GLM
LogLike	622.5935	622.5935
int	-0.0819	-0.0818
review_year_0	0.0213	0.0213
age_sc	-0.2213	-0.2213
word_count_sc	-0.7344	-0.7343

So when it comes to classification, you should feel confident in what's going on under the hood, just like you did with a numeric target. Conceptually it really is the same approach.

## 4.11 Optimization Algorithms

When it comes to optimization, there are a number of algorithms that have been developed over time. We'll demonstrate one of the most popular ones used in machine learning, but there many variants of this one even! The main thing to keep in mind is that these are all just ways to find the best fitting parameters for a model. The algorithms differ in how they do this, and some may be better suited for certain data tasks, or provide computational advantages.

### 4.11.1 Gradient Descent

One of the most common approaches in optimization is called **gradient descent**. The idea behind it is that we can use the gradient of the objective function to guide us to the best fitting parameters. We still use estimation approaches like maximum likelihood - gradient descent is just a way to find that path along the objective surface. More formally, the gradient is the vector of partial derivatives of the objective function with respect to each parameter. That may not mean much to you, but the basic idea is that the gradient is a vector that points in the direction of steepest ascent in terms of the objective function. So if we want to maximize the objective function, we can take a step in the direction of the gradient, and if we want to minimize it, we can take a step in the opposite direction of the gradient. The size of the step is called

the **learning rate**, and, like our penalty parameter we saw with penalized regression, it is a hyperparameter that we can tune. If the learning rate is too small, it will take a longer time to converge. If the learning rate is too large, we might overshoot the objective and never converge. There are a number of variations on gradient descent that have been developed over time. Here is a function to illustrate the process. Let's see this in action with the happiness data model we used previously.

#### 4.11.1.1 R

```
gradient_descent = function(
  par,
  X,
  y,
  tolerance = 1e-3,
  maxit = 1000,
  learning_rate = 1e-3,
  adapt = FALSE,
  verbose = TRUE
) {
  # add a column of 1s for the intercept
  X = cbind(1, X)
  N = nrow(X)

  # initialize
  beta = par
  names(beta) = colnames(X)
  mse = crossprod(X %*% beta - y) / N
  tol = 1
  iter = 1

  while (tol > tolerance && iter < maxit) {
    LP = X %*% beta
    grad = t(X) %*% (LP - y)
    betaCurrent = beta - learning_rate * grad
    tol = max(abs(betaCurrent - beta))
    beta = betaCurrent
    mse = append(mse, crossprod(LP - y) / N)
    iter = iter + 1

    if (adapt) {
      stepsize = ifelse(
        mse[iter] < mse[iter - 1],
        stepsize * 1.2,
        stepsize * .8
    }
  }
}
```

```

        )
}

if (verbose && iter %% 10 == 0) {
  message(paste("Iteration:", iter))
}
}

list(
  par    = beta,
  loss   = mse,
  MSE    = crossprod(LP - y) / nrow(X),
  iter   = iter,
  fitted = LP
)
}

our_result = gradient_descent(
  par = c(0, 0, 0, 0),
  X = df_happiness |> select(life_exp, gdp_pc, corrupt) |> as.matrix(),
  y = df_happiness$happiness,
  learning_rate = 1e-3,
  verbose = FALSE
)

```

#### 4.11.1.2 Python

```

def gradient_descent(
  par,
  X,
  y,
  tolerance = 1e-3,
  maxit = 1000,
  learning_rate = 1e-3,
  adapt = False,
  verbose = True
):
  # add a column of 1s for the intercept
  X = np.c_[np.ones(X.shape[0]), X]

  # initialize
  beta = par
  loss = np.sum((X @ beta - y)**2)
  tol = 1
  iter = 1

```

```

while (tol > tolerance and iter < maxit):
    LP = X @ beta
    grad = X.T @ (LP - y)
    betaCurrent = beta - learning_rate * grad
    tol = np.max(np.abs(betaCurrent - beta))
    beta = betaCurrent
    loss = np.append(loss, np.sum((LP - y)**2))
    iter = iter + 1

    if (adapt):
        stepsize = np.where(
            loss[iter] < loss[iter - 1],
            stepsize * 1.2,
            stepsize * .8
        )

    if (verbose and iter % 10 == 0):
        print("Iteration:", iter)

return({
    "par": beta,
    "loss": loss,
    "RSE": np.sqrt(np.sum((LP - y)**2) / (X.shape[0] - X.shape[1])),
    "iter": iter,
    "fitted": LP
})

our_result = gradient_descent(
    par = np.array([0, 0, 0, 0]),
    X = df_happiness[['life_exp', 'gdp_pc', 'corrupt']].to_numpy(),
    y = df_happiness['happiness'].to_numpy(),
    learning_rate = 1e-3,
    verbose = False
)

```

Comparing our results, we have the following table. In what has become the norm, we see that the results are very similar.

Table 4.7: Comparison of gradient descent results

Value	Built-in	Our Result
Intercept	5.445	5.437
Life Exp. Coef.	0.525	0.521
GDP_PC	0.438	0.439

Corrupt	-0.105	-0.107
MSE	0.367	0.367

In addition, when we visualize the loss function across iterations, we see smooth decline in the MSE value as we go along each iteration. This is a good sign that we are converging to a good solution.

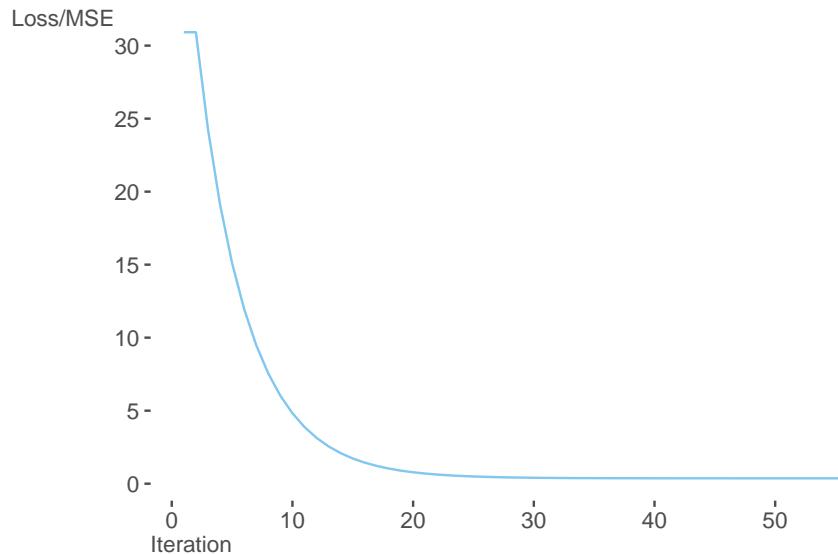


Figure 4.7: Gradient descent path

#### 4.11.2 Stochastic Gradient Descent

**Stochastic gradient descent** (SGD) is a variation on gradient descent that uses a random sample of the data to estimate the gradient, while the ‘true’ gradient is the gradient of the objective function with respect to all of the data. As such, it’s less accurate than the ‘batch’ gradient descent in some sense, but the advantage of SGD is that it is faster. In practice, SGD is often used in machine learning applications where the data is large, and the tradeoff between accuracy and speed is worth it.

Let’s see this in action with the happiness data model we used previously. The following is a conceptual version of the AdaGrad approach<sup>9</sup>, which is a variation of stochastic gradient descent that adjusts the learning rate for each parameter. We will also add a variation that averages the parameter estimates across iterations, which is a common approach to improve the performance

---

<sup>9</sup>MC does not recall exactly where this origin of his function came from except that Murphy’s PML book was a key reference (Murphy (2012)).

of stochastic gradient descent, but by default it is not used, just something you can play with. We are going to use a ‘batch size’ of one, which is similar to a ‘streaming’ or ‘online’ version where we update the model with each observation. Since our data are alphabetically ordered, we’ll shuffle the data first. We’ll also use a stepsize\_tau parameter, which is a way to adjust the learning rate at early iterations. We’ll set it to zero for now, but you can play with it to see how it affects the results. The values for the learning rate and stepsize\_tau are arbitrary, selected after some initial playing around, but you can play with them to see how they affect the results.

TODO: SHOULD MAYBE CLEAN UP/ALTER TO LESS VERBOSE VERSION

#### 4.11.2.1 R

```
stochastic_gradient_descent = function(
  par, # parameter estimates
  X, # model matrix
  y, # target variable
  learning_rate = 1, # the learning rate
  stepsize_tau = 0, # if > 0, a check on the LR at early iterations
  average = FALSE # a variation of the approach
) {
  # initialize
  X = cbind(1, X)
  beta = par

  # Collect all estimates
  betamat = matrix(0, nrow(X), ncol = length(beta))

  # Collect fitted values at each point))
  fits = NA

  # Collect loss at each point
  loss = NA

  # adagrad per parameter learning rate adjustment
  s = 0

  # a smoothing term to avoid division by zero
  eps = 1e-8

  for (i in 1:nrow(X)) {
    Xi = X[i, , drop = FALSE]
    yi = y[i]
```

```

# matrix operations not necessary here,
# but makes consistent with standard gd func
LP = Xi %*% beta
grad = t(Xi) %*% (LP - yi)
s = s + grad^2 # adagrad approach

# update
beta = beta - learning_rate / (stepsize_tau + sqrt(s + eps)) * grad

# a variation
if (average & i > 1) {
  beta = beta - 1 / i * (betamat[i - 1, ] - beta)
}

betamat[i, ] = beta
fits[i] = LP
loss[i] = crossprod(LP - yi)
}

LP = X %*% beta
lastloss = crossprod(LP - y)

list(
  par = beta, # final estimates
  par_chain = betamat, # estimates at each iteration
  MSE = sum(lastloss) / nrow(X),
  fitted = LP
)
}

# setting a seed ensures replicability
set.seed(123)

# generate random sample indices (could also have done within the function)
idx = sample(1:nrow(df_happiness), nrow(df_happiness))

X_train = df_happiness |>
  select(life_exp, gdp_pc, corrupt) |>
  dplyr::slice(idx) |>
  as.matrix()

y_train = df_happiness$happiness[idx]

our_result = stochastic_gradient_descent(
  par = c(mean(df_happiness$happiness), 0, 0, 0),

```

```

X = X_train,
y = y_train,
learning_rate = .15,
stepsize_tau = .1
)

```

#### 4.11.2.2 Python

```

def stochastic_gradient_descent(
    par, # parameter estimates
    X, # model matrix
    y, # target variable
    learning_rate = 1, # the learning rate
    stepsize_tau = 0, # if > 0, a check on the LR at early iterations
    average = False # a variation of the approach
):
    # initialize
    X = np.c_[np.ones(X.shape[0]), X]
    beta = par

    # Collect all estimates
    betamat = np.zeros((X.shape[0], beta.shape[0]))

    # Collect fitted values at each point))
    fits = np.zeros(X.shape[0])

    # Collect loss at each point
    loss = np.zeros(X.shape[0])

    # adagrad per parameter learning rate adjustment
    s = 0

    # a smoothing term to avoid division by zero
    eps = 1e-8

    for i in range(X.shape[0]):
        Xi = X[None, i, :]
        yi = y[i]

        # matrix operations not necessary here,
        # but makes consistent with standard gd func
        LP = Xi @ beta
        grad = Xi.T @ (LP - yi)
        s = s + grad**2 # adagrad approach

```

```

# update
beta = beta - learning_rate / \
      (stepsize_tau + np.sqrt(s + eps)) * grad

# a variation
if (average & i > 1):
    beta = beta - 1 / i * (betamat[i - 1, :] - beta)

betamat[i, :] = beta
fits[i] = LP
loss[i] = np.sum((LP - yi)**2)

LP = X @ beta
lastloss = np.sum((LP - y)**2)

return({
    "par": beta, # final estimates
    "par_chain": betamat, # estimates at each iteration
    "MSE": lastloss / X.shape[0],
    "fitted": LP
})

# setting a seed ensures replicability
np.random.seed(1234)

# generate random sample indices (could also have done within the function)
idx = np.random.choice(
    df_happiness.shape[0],
    df_happiness.shape[0],
    replace = False
)

X_train = df_happiness[['life_exp', 'gdp_pc', 'corrupt']].to_numpy()[idx, :]
y_train = df_happiness['happiness'].to_numpy()[idx]

our_result = stochastic_gradient_descent(
    par = np.array([np.mean(df_happiness['happiness']), 0, 0, 0]),
    X = X_train,
    y = y_train,
    learning_rate = .15,
    stepsize_tau = .1
)

```

Next we'll compare it to OLS estimates. Very similar even though SGD nor-

mally would not be used for such a small dataset. We also show our previous ‘batch’ gradient descent results for comparison.

Table 4.8: Comparison of stochastic gradient descent results

Value	Built-in	Our Result	Batch SGD
Intercept	5.445	5.469	5.437
Life Exp. Coef.	0.525	0.514	0.521
GDP_PC	0.438	0.390	0.439
Corrupt	-0.105	-0.111	-0.107
MSE	0.367	0.370	0.367

And here’s a plot of the estimates as they moved along the data. For this plot we don’t include the intercept as it’s on a notably different scale. We can see that the estimates are moving around a bit, but they appear to be converging to a solution.

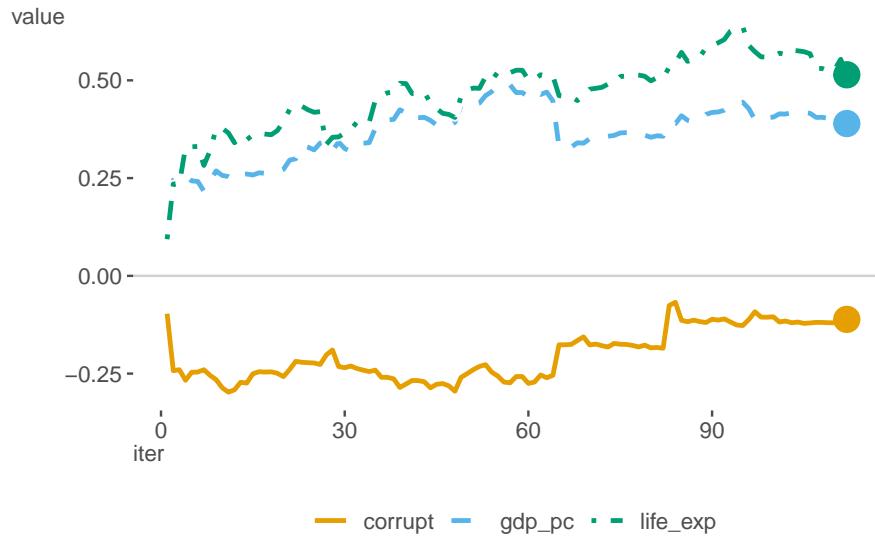


Figure 4.8: Stochastic gradient descent path

#### 4.11.3 Other Optimization Algorithms

There are lots of other approaches to optimization. For example, here are some of the options available in R’s `optim` or `scipy`’s `minimize` function:

- Nelder-Mead
- BFGS

- L-BFGS-B (provides constraints)
- Conjugate gradient
- Simulated annealing
- Newton's method
- Genetic algorithms

The main reason to choose one method over another usually is some sort of computational gain, e.g. memory or speed, or it may just work better for some types of models in practice. For statistical problems, many GLM-type functions appear to use Newton's as a default, but more complicated models may implement a different default for better convergence. In general, we can always try a few different methods to see which works best, and often there would be little differences in the results. For example, here are the results for the happiness model using different algorithms, with a comparison to the standard linear regression model function. We can see that the results are very similar, and for simpler modeling endeavors they should converge on the same result.

Table 4.9: Comparison of optimization results

parameter	NM <sup>1</sup>	BFGS <sup>2</sup>	CG <sup>3</sup>	GD <sup>4</sup>	Built-in <sup>5</sup>
Intercept	5.445	5.445	5.445	5.437	5.445
Life Exp. Coef.	0.525	0.525	0.525	0.521	0.525
GDP_PC	0.437	0.438	0.438	0.439	0.438
Corrupt	-0.105	-0.105	-0.105	-0.107	-0.105
MSE	0.367	0.367	0.367	0.367	0.367

<sup>1</sup>NM = Nelder-Mead

<sup>2</sup>BFGS = Broyden–Fletcher–Goldfarb–Shanno

<sup>3</sup>CG = Conjugate gradient

<sup>4</sup>GD = Gradient descent

<sup>5</sup>Built-In = Standard OLS function

---

## 4.12 Other Estimation Approaches

Before leaving our estimation discussion, we should mention that there are other approaches to estimation that are out there, some quite common. These include variations on least squares, **method of moments**, **generalized estimating equations**, **robust** estimation, and more. The above that we've focused on will generally be sufficient for most applications, but it's good to be aware of others. But there are two we want to discuss in a little bit detail before we leave model estimation formally given their widespread usage, and that is the **bootstrap** and **Bayesian estimation**.

### 4.12.1 Bootstrap

The **bootstrap** is a resampling approach to estimation. We **sample with replacement** from the data observations, generating an entirely new data set of the same size, and then estimate the model. We repeat this process many times, collecting parameter estimates, predictions, or any thing we want to calculate along the way. Ultimately we end up with a distribution of possible parameter estimates, metrics, and whatever else we calculated.

This distribution is useful for **inference**<sup>10</sup>, as we can use the distribution to calculate confidence intervals, prediction intervals or intervals for anything we happen to calculate. The average estimate will typically be the same as whatever the underlying model used would produce, but the bootstrap provides a way to get at a measure of **uncertainty** with fewer assumptions about how that distribution should take shape. The bootstrap is very flexible, and it can be used with any estimation approach, let's see this in action with the happiness data model we used previously.

#### 4.12.1.1 R

```
bootstrap = function(X, y, nboot = 100, seed = 123) {
  # add a column of 1s for the intercept
  N = nrow(X)

  # initialize
  beta = matrix(NA, (1+ncol(X))*nboot, nrow = nboot, ncol = 1+ncol(X))
  colnames(beta) = c('Intercept', colnames(X))
  mse = rep(NA, nboot)

  # set seed
  set.seed(seed)

  for (i in 1:nboot) {
    # sample with replacement
    idx = sample(1:N, N, replace = TRUE)
    Xi = X |> slice(idx)
    yi = y[idx]

    # estimate model
    mod = lm(yi ~., data = Xi)

    # save results
    beta[i, ] = coef(mod)
```

---

<sup>10</sup>We're using inference here in the statistical/philosophical sense, not as a synonym for prediction or generalization, which is how it is often used in machine learning. We're not exactly sure how that terminological muddling arose in ML, but be on the lookout for it.

```

    mse[i] = sum((mod$fitted - yi)^2) / N
}

# given mean estimates, calculate MSE
y_hat = cbind(1, as.matrix(X)) %*% colMeans(beta)
final_mse = sum((y - y_hat)^2) / N

list(
  beta = as_tibble(beta),
  MSE = mse,
  final_mse = final_mse
)
}

our_result = bootstrap(
  X = df_happiness |> select(life_exp, gdp_pc, corrupt),
  y = df_happiness$happiness,
  nboot = 250
)

```

#### 4.12.1.2 Python

```

def bootstrap(X, y, nboot=100, seed=123):
    cn = X.columns
    # add a column of 1s for the intercept
    X = np.c_[np.ones(X.shape[0]), X]
    N = X.shape[0]

    # initialize
    beta = np.empty((nboot, X.shape[1]))

    # beta = pd.DataFrame(beta, columns=['Intercept'] + list(cn))
    mse = np.empty(nboot)

    # set seed
    np.random.seed(seed)

    for i in range(nboot):
        # sample with replacement
        idx = np.random.randint(0, N, N)
        Xi = X[idx, :]
        yi = y[idx]

        # estimate model
        model = LinearRegression(fit_intercept=False)

```

```

mod = model.fit(Xi, yi)

# save results
beta[i, :] = mod.coef_
mse[i] = np.sum((mod.predict(Xi) - yi)**2) / N

# given mean estimates, calculate MSE
y_hat = X @ beta.mean(axis=0)
final_mse = np.sum((y - y_hat)**2) / N

return dict(beta = beta, mse = mse, final_mse = final_mse)

our_result = bootstrap(
    X = df_happiness[['life_exp', 'gdp_pc', 'corrupt']],
    y = df_happiness['happiness'],
    nboot = 250
)

```

Here are the results of the interval estimates for the coefficients. For each parameter, we have the mean estimate, the lower and upper bounds of the 95% confidence interval, and the width of the interval. We can see that the bootstrap intervals are wider than the OLS intervals, possibly better capturing the uncertainty in this model based on not too many observations.

Table 4.10: Bootstrap parameter estimates

Parameter	mean	Lower BS	Upper BS	Lower OLS	Upper OLS	Diff Width <sup>1</sup>
Intercept	5.44	5.33	5.55	5.33	5.56	-0.02
life_exp	0.51	0.30	0.70	0.35	0.70	0.04
gdp_pc	0.46	0.18	0.76	0.24	0.64	0.18
corrupt	-0.10	-0.29	0.09	-0.25	0.04	0.09

<sup>1</sup>Width of bootstrap estimate minus width of OLS estimate

Let's look more closely at the bootstrap distributions for each coefficient. With standard statistical estimates, we are assuming a distribution like the normal, which is a very specific shape. With the bootstrap, we can be more flexible, though often it may tend toward the distribution that would otherwise be assumed anyway. These aren't perfectly symmetrical, but they suit our needs in that we can extract the lower and upper quantiles to create an interval estimate.

The bootstrap is a commonly used for predictions and other metrics, but it is computationally inefficient, and can become prohibitive with large data sizes. Also, the simple bootstrap will likely not estimate the appropriate uncertainty

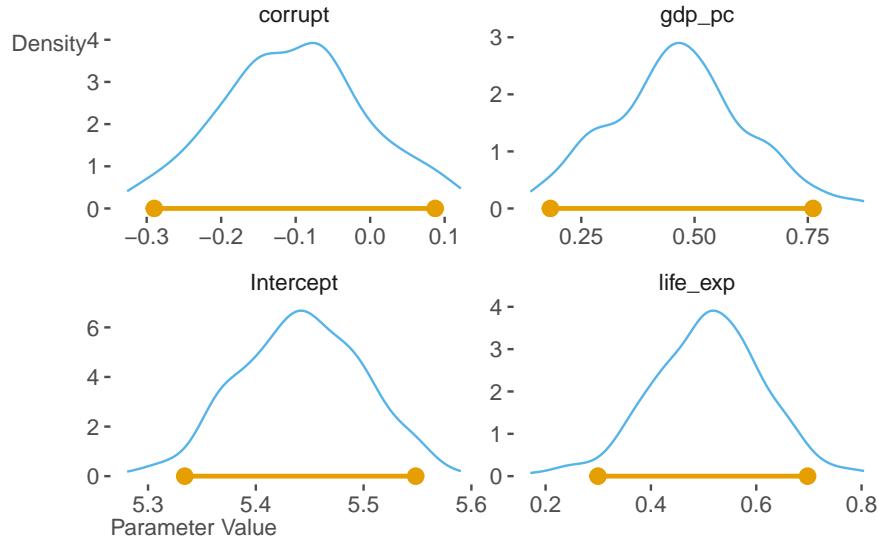


Figure 4.9: Bootstrap distributions of parameter estimates

for some types of statistics (e.g. extreme values) or in some data contexts<sup>11</sup> (e.g. correlated observations). Overcoming the limitations may typically require an even more computationally intensive approach, further limiting its utility. But it is a useful tool to have in your toolbox, and it can be used in conjunction with other approaches to get at uncertainty in a model.

#### 4.12.2 Bayesian Estimation

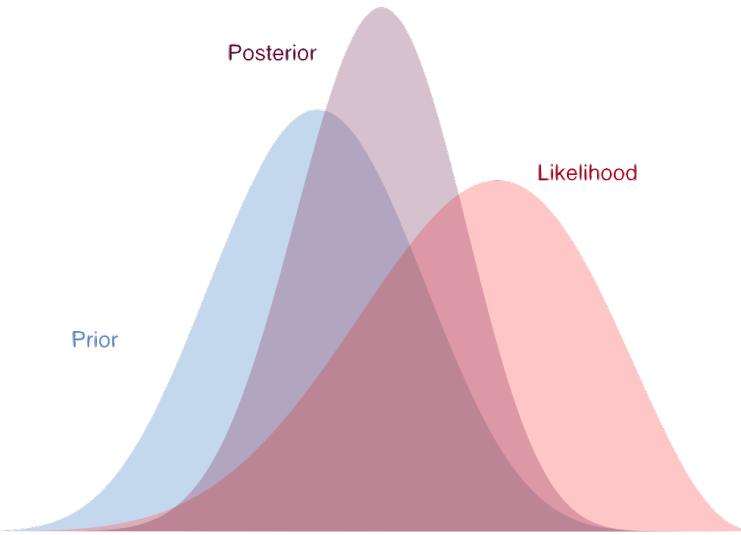
The **Bayesian** approach to modeling is a philosophical viewpoint, an entirely different way to think about probability, a different way to measure uncertainty, and on a practical level, just another way to get model parameter estimates. It can be as frustrating as it is fun to use, and one of the really nice things about using Bayesian estimation is that it can handle model complexities that other approaches don't do well.

The basis of Bayesian estimation is the **likelihood**, the same as used with maximum likelihood, and everything we did there follows to here. However, here we can incorporate domain knowledge about the parameters, in the form of **prior distributions**, which we specify in addition to the likelihood. For example, we may say that the coefficients for a linear model come from a normal distribution centered on zero with some variance. The combination prior distributions with the likelihood ultimately results in the **posterior distri-**

---

<sup>11</sup><https://stats.stackexchange.com/questions/9664/what-are-examples-where-a-naive-bootstrap-fails>

**bution.** And this is the key difference when comparing Bayesian estimation to the others we've talked about, and something it shares in common with the bootstrap- the end result is not a point estimate of the parameters, but rather a *distribution* of possible parameter values.



Dealing with distributions instead of single estimates is a different way to think about modeling, but it can be very useful. For example, as we did with the bootstrap, the Bayesian posterior distribution is useful for inference. With these distributions, we can look at any range in between for our **credible interval**, which is the Bayesian equivalent of a confidence interval<sup>12</sup>. Here is an example of the posterior distribution for the parameters of our happiness model, along with 95% intervals.

With Bayesian modeling, we use the algorithm of our choosing, give it starting values and proceed much in the same way as other optimization procedures. However, in this approach, we always specify a number of iterations as the **stopping rule**, i.e. when the model should terminate. These iterations are single draws from the posterior distribution for each parameter. So if we specified 1000 iterations, we would have 1000 draws from the posterior distribution for each parameter. Typically we don't use the first few hundred draws, as these are considered **burn-in** or **warmup** draws, and we use the remaining draws for inference. The number of burn-in draws is a bit of an art, but it's not too important as long as it's not too small. The more iterations we set,

<sup>12</sup>Your default interpretation of a standard confidence interval is almost certainly, and incorrectly, the actual interpretation of a Bayesian confidence interval, because the Bayesian interpretation of confidence intervals and p-values is how we tend to naturally think about them. But that's okay, everyone else is in the same boat. We also don't care if you want to call the Bayesian version a credible interval or a confidence interval.

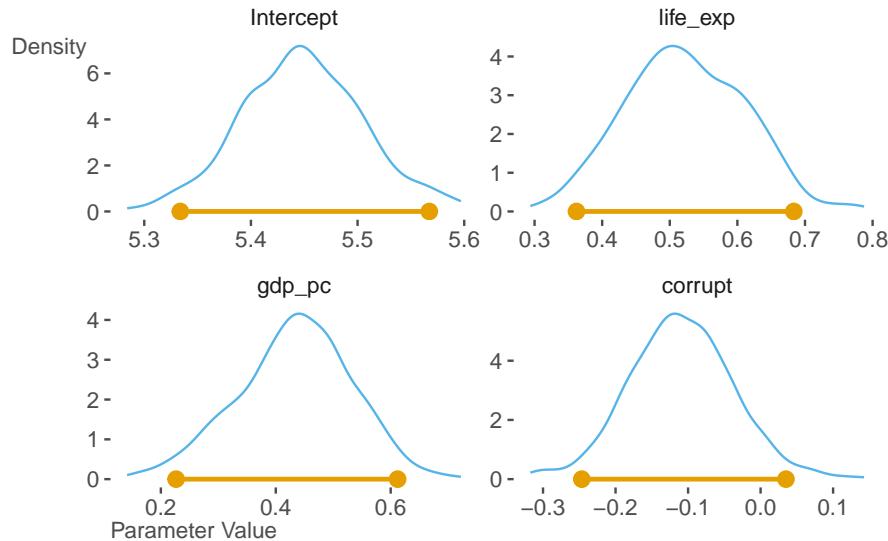


Figure 4.10: Posterior distribution of parameters

the longer it will take to run. We also specify multiple **chains**, which are each doing the exact same thing, but do to the random nature of the Bayesian approach, would take different estimation paths. We can then compare the chains to see if they are converging to the same result, which is a check on the model. If they are not converging, we may need to run the model longer, or we may need to change something else. Here is an example of the chains for our happiness model for the life expectancy coefficient. We can see that they are converging to the same result, so we are good to go. Nowadays we have simple metrics that allow us to check whether the chains are converging, making it easier to assess many parameters quickly.

When we are interested in making predictions, we can use the results to generate a distribution of possible predictions *for each observation*, which can be very useful when we want to quantify uncertainty in for complex models. This is referred to as **posterior predictive distribution**. Here is a plot of several draws of predicted values against the true happiness scores.

Note that *any* metric we can calculate from a model will also have a distribution. For example, you have a classification model and you want to know the accuracy or true positive rate of the model. Instead of a single number, you now have access to a whole distribution of values for that metric. Why? For every sample of the distribution of parameters, you generate a prediction, convert it a class and compare it to the true class. So now you have a posterior predictive distribution for the predicted probabilities and class, and you can then calculate the accuracy, area under a receiver operating curve, true posi-

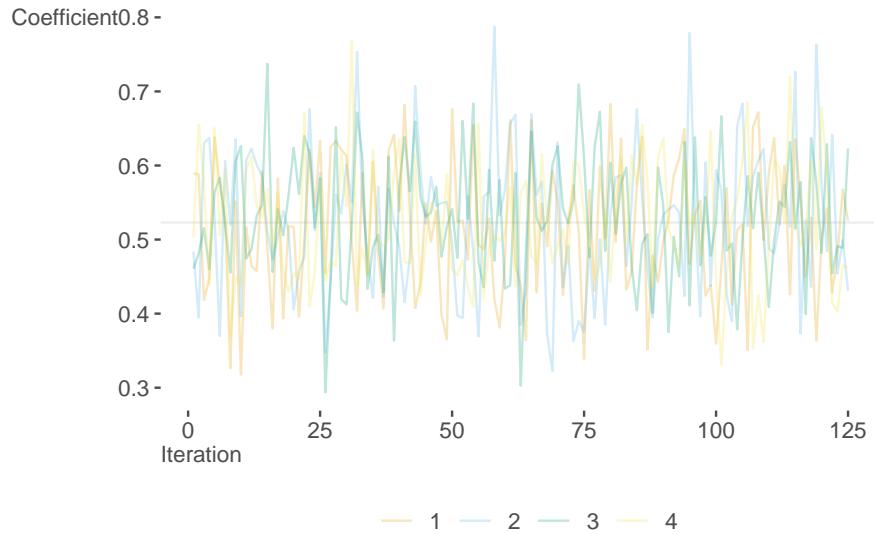


Figure 4.11: Bayesian chains for life expectancy coefficient

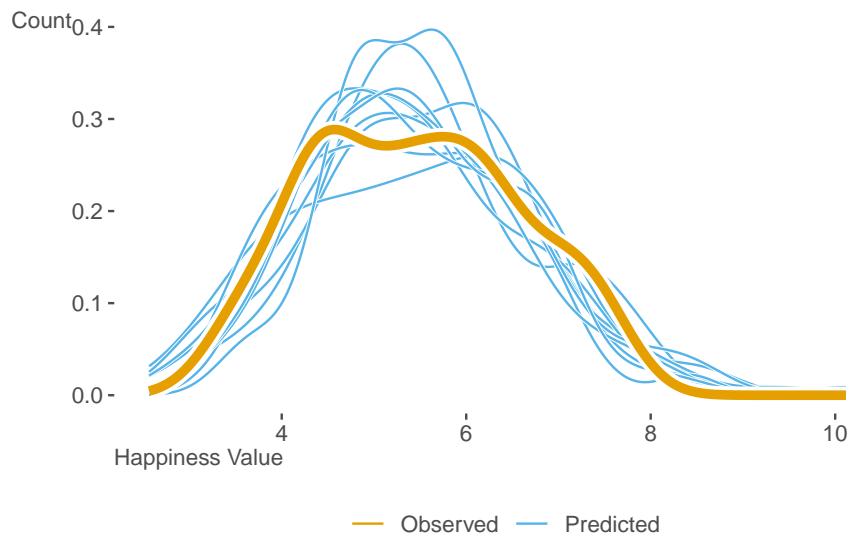


Figure 4.12: Posterior predictive distribution of happiness values

tive rate, etc., for each sample, and you have a distribution of possible values. As an example, we did this for our happiness model and show the interval estimate for R-squared. Pretty neat!

Table 4.11: Bayesian R<sup>2</sup>

Bayes R2	Lower	Upper
0.71	0.65	0.75

95% Credible interval for R-squared

💡 There is nothing keeping you from doing posterior predictive checks with other estimation approaches, and it's a good idea to do so. For example, in a GLM you have the beta estimates and the covariance matrix for them, and can simulate from a normal distribution with those estimates. But it's a bit more straightforward with the Bayesian approach, and some packages will allow you to do this automatically even.

#### 4.12.2.1 Additional Thoughts

It turns out that any standard (frequentist) statistical model can be seen as a Bayesian one from a particular point of view. Here are a couple:

- GLM and related estimated via maximum likelihood: Bayesian estimation with a flat/uniform prior on the parameters.
- Ridge Regression: Bayesian estimation with a normal prior on the coefficients, penalty parameter is related to the variance of the prior
- Lasso Regression: Bayesian estimation with a Laplace prior on the coefficients, penalty parameter is related to the variance of the prior

So in many modeling contexts, you're actually doing a restrictive form of Bayesian estimation already. Hopefully this helps to demystify the Bayesian approach a bit, and you feel more comfortable switching to it. R has excellent tools here for modeling and post-processing, like `brms` and `tidybayes`, and Python has `pymc3`, `numpyro`, and `arviz`, which are also useful<sup>13</sup>.

We can see that the Bayesian approach is very flexible, and can be used for many different types of models, and can be used to get at uncertainty in a model in ways that other approaches can't. It's not a panacea, and it's not always the best approach, but it's a good one to have in your toolbox.

---

<sup>13</sup>Honestly R has way more going on here, with many packages devoted to Bayesian estimation of specific models even, but if you want to stick with Python for it you at least have some options. Stan, a probabilistic language underlying many of the packages in R, has tools there as well, but they are not nearly as well developed or test in Python.

TODO: WHERE TO PUT THIS PRIOR STUFF? DELETE!?

The tough part about the Bayesian approach is specifying priors, but even when you don't have a great idea, many have offered solutions, and there are ways to check whether what you've chosen makes sense for your data before trying the model itself.

💡 Specification of priors can be done in different ways, and nowadays, there is a lot of information on how to do so, and with some tools, it's also pretty straightforward to check whether the priors are sensible without even running a model. When you do have actual prior knowledge, either domain knowledge (e.g. a prior study found the beta values to be positive), statistical knowledge, (e.g. only the largest standard coefficients go near or beyond 1), data from time periods, there's typically at least something to help you specify your priors with sensible values. This takes away most of the luster of the primary argument against the Bayesian approach, which is the subjective nature of priors. But there is likewise so much subjective decision making in other approaches, that it's not really a useful argument to begin with. The Bayesian approach just makes it more explicit. And if you don't have any prior knowledge, you can use non- or weakly-informative priors, which will likely have little influence and let the data do the talking, producing a result that is not that different from maximum likelihood estimation.

### 4.13 Wrapping Up

Wow, we covered a lot here! But this is the sort of stuff that can take you from just having some fun with data, to doing that and also understanding how things are actually happening. Just having the gist of how modeling actually is done ‘under the hood’ makes so many other things make sense, and can give you a lot of confidence, even in less familiar modeling domains.

### 4.14 Where to Go From Here

Really, after this chapter, you should feel fine with any of the others, so dive in! Here are some additional resources to consider.

## 4.15 Exercise

Try creating an objective function for a continuous target that uses the mean absolute error, and compare your estimated parameters to the previous results.

TODO: refs need work!

### OLS and Maximum Likelihood Estimation:

For OLS and maximum likelihood estimation, there are so many resources out there, we recommend just taking a look and seeing which one suits you best. Practically any more technical statistical book will cover these topics in detail.

- A list of classical references<sup>14</sup>
- TODO: recent?

### Gradient Descent:

- Gradient Descent, Step-by-Step<sup>15</sup> StatQuest with Josh Starmer (2019a)
- Stochastic Gradient Descent, Clearly Explained<sup>16</sup> StatQuest with Josh Starmer (2019b)

The simple AdaGrad algorithm used above:

- Brownlee (2021)
- DataBricks (2019)

### Bootstrap: ?

### Bayesian:

- BDA Gelman et al. (2013)
- Statistical Rethinking McElreath (2020)
- Choosing priors<sup>17</sup>

---

<sup>14</sup><https://stats.stackexchange.com/questions/33197/advanced-statistics-books-recommendation>

<sup>15</sup><https://www.youtube.com/watch?v=sDv4f4s2SB8>

<sup>16</sup><https://www.youtube.com/watch?v=vMh0zPT0tLI>

<sup>17</sup><https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations>



# 5

---

## *Generalized Linear Models*

---

What happens when your target variable isn't really a continuous variable, but is instead some other type of response. Maybe you've got a binary condition, like good or bad, or maybe you've got a count of something, like the number of times a person has been arrested. In these cases, you can use a linear regression, but it often won't do exactly what you want. But you can use a **generalized linear model** to help yourself in these situations.

Generalized linear models exist to map different distributions into linear space. This allows us to use the same linear model framework that we've been using, but with different types of data.

These models work by **generalizing** the linear model to different distributions of the target variable. Our coefficients will certainly take on a new meaning; so while we cannot interpret them as we would coefficients from a linear regression, we can still use the general framework.

---

### 5.1 Key Ideas

- A simple tweak to our previous approach allows us to generalize a linear model to account for other settings.
- Common distributions such as binomial, poisson, and others can often do better for us both in terms of model fit and interpretability.
- Getting familiar with just a couple distributions will allow you to really expand your modeling repertoire.

#### 5.1.1 Why this matters

The linear model is powerful on its own, but even more so when you realize you can extend many other data settings, some of which are implicitly nonlinear! When we want to classify observations, count them, or deal with proportions and other things, simple tweaks of our standard linear model allow us to handle such situations.

### 5.1.2 Good to know

Generalized linear models are a broad class of models that extend the linear model to different distributions of the target variable. In general, you'd need to have a pretty good grasp of linear regression before getting too carried away here.

## 5.2 Distributions & Link Functions

Remember how linear models really enjoy the whole Gaussian distribution scene? The essential form of the linear model can be expressed as follows:

$$\begin{aligned}\mu &= \alpha + X\beta \\ y &\sim \text{Normal}(\mu, \sigma)\end{aligned}$$

Not all data follows a Gaussian distribution. Instead, we often find some other form of an exponential distribution. So, we need a way to incorporate different distributions of the target into our model. Distributions cannot do it alone! We also need a **link function** to connect the linear model to the distribution.

From a theoretical perspective, link functions are tricky to get your head around.

- *Find the exponential of the response's density function and derive the canonical link function...*

From a conceptual perspective, all they are doing is allowing the linear feature to “link” to a distribution function’s mean. If you know a distribution’s canonical link function, that is all the deeper you will probably every need.

At the end of the day, these link functions will convert the target to an unbounded continuous variable. The take-away here is that the link function describes how the mean is generated from the predictors.

## 5.3 Logistic Regression

### 5.3.1 Why Should You Care

You will often have a binary variable that you might want to use as a target – it could be dead/alive, lose/win, quit/retain, etc. You might be tempted to use a linear regression, but you will quickly find that it is not the best option.

You are going to be figuring out the probability of moving from “failure” to “success”, given the features in your model.

### 5.3.2 The Binomial Distribution

Logistic regression is substantially different than linear regression. It is also a bit confusing, because it is named after its link function (**logit**) instead of its distribution (**binomial**). Instead of that nice continuous target, we are dealing with a binomially-distributed target and the target takes the form of a binary variable.

We don’t have a  $\mu$  or  $\sigma^2$  to identify the shape of the binomial distribution; instead we have  $p$  and  $n$ , where  $p$  is a probability and  $n$  is the number of trials. We tend to talk about  $p$  with regard to the probability of a specific event happening (heads, wins, defaulting, etc.).

Let’s see how the binomial distribution looks with 100 trials and probabilities of “success” at  $p = .25, .5$ , and  $.75$ :

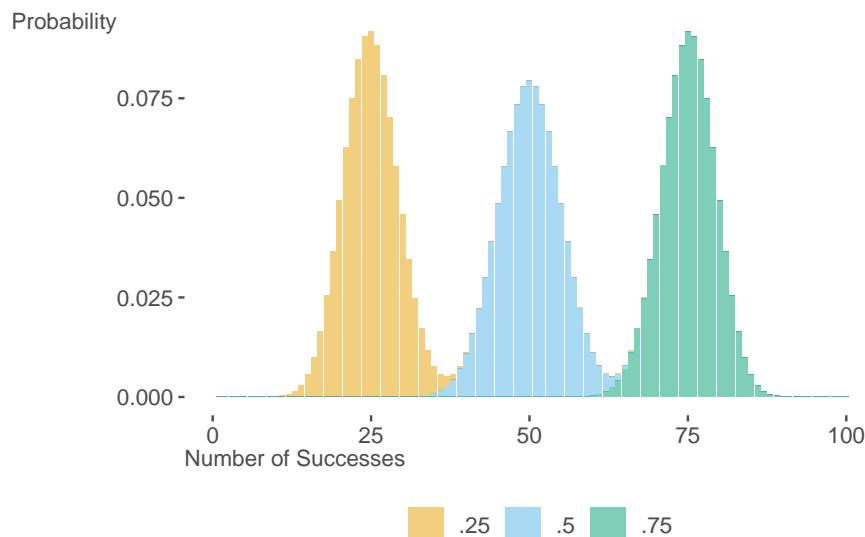


Figure 5.1: Binomial distributions for different probabilities

If we examine the distribution for a probability of  $.5$  in Figure 5.1, we will see that it is centered over  $50$  – this would suggest that we have the highest probability of encountering  $50$  successes if we ran  $100$  trials. If we run  $100$  trials  $100$  times and the outcome is  $50/50$ , the most common outcome from those  $100$  trials would be  $50$  successes, with a decreasing probability of observing more or less successes as we move away from  $50$ . Shifting our attention to a  $.75$

probability of success, we see that our density is sitting over 75. Again running 100 trials, would give us the highest probability of observing 75 successes. Some of those 100 trials produce more or less than 75 successes, but with lower probabilities as you get further away from 75.

Since we are dealing with a number of trials, it is worth noting that the binomial distribution is a discrete distribution. If you have any interest in knowing the probability for a number of success under the binomial distribution, we can use the following formula:

$$P(x) = \frac{n!}{(n-x)!x!} p^x q^{n-x}$$

While we don't need to dive into finding those specific values for the binomial distribution, we can spend our time exploring how it looks in linear model space:

$$\text{logit}(p) = \alpha + X\beta$$

$$y \sim \text{Binomial}(n, p)$$

The *logit* function is defined as:

$$\log \frac{p}{1-p}$$

We are literally just taking the log of the odds (the log odds becomes important later).

Now we can map this back to our model:

$$\log \frac{p}{1-p} = \alpha + X\beta$$

And finally we can take that logistic function and invert it (the **inverse-logit**) to produce the probabilities.

$$p = \frac{\exp(\alpha + X\beta)}{1 + \exp(\alpha + X\beta)}$$

Whenever we get coefficients for the logistic regression model, we are always going to get them as log odds. We can exponentiate them to get the odds ratio, but we can also exponentiate them and divide by 1 + that value to get the probability.

### 5.3.3 Probability, Odds, and Log Odds

Probability lies at the heart of all of this. We can look at the relationship between the probability, odds, and log odds. We can start with a set of probability values where  $0 < p > 1$

With that list of probability values, we can convert them to odds with  $p / 1 - p$ .

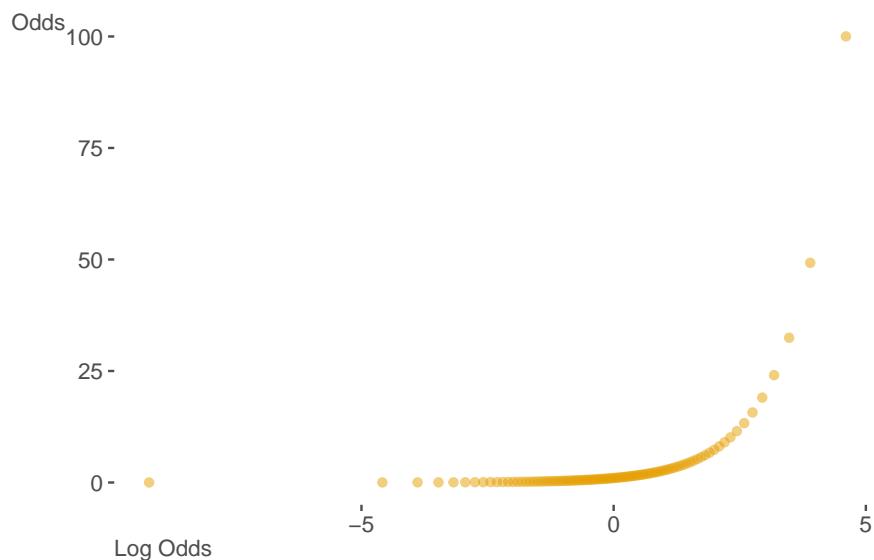


Figure 5.2: Log odds and odds values for a range of probabilities

We can see how those probability values map to odds in Figure 5.2.

Now, we can take those odds values and convert them to log odds.

If you've ever seen the sigmoid featured in Figure 5.3 before, it is the classic logistic function!

We can clearly go back and forth between the 3, but the main message here is that we took a bounded variable in probability and transformed it to continuous space.

We will see more about how this happens after playing with the model.

### 5.3.4 Data Import and Preparation

We are going to return to our movie reviews data and we are going to use `rating_good` as our target. Before we get to modeling, see if you can find out the frequency of “good” and “bad” reviews. We will use `word_count` and `gender`

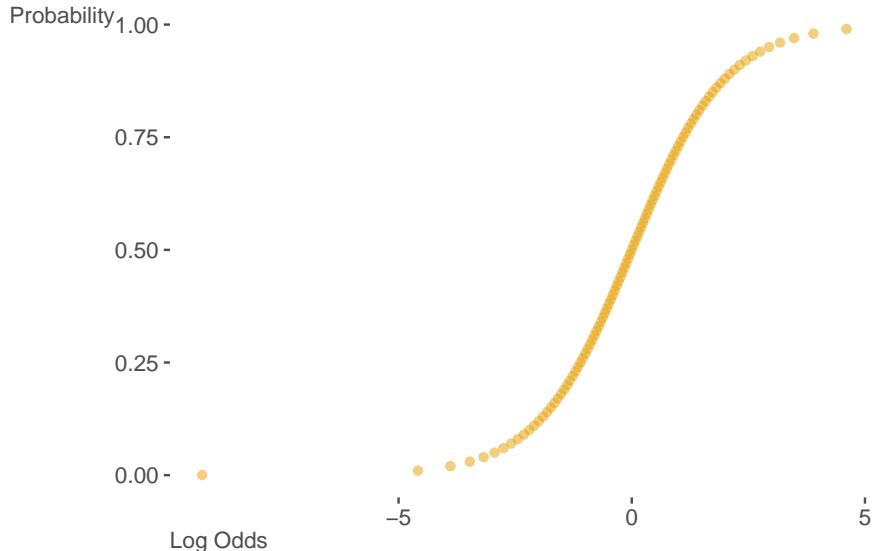


Figure 5.3: Log odds and probability values

as our predictors. Before we move on, though, find the probability of getting a “good” review.

#### 5.3.4.1 R

```
reviews = read.csv("data/movie_reviews_processed.csv")
X = reviews[, c("word_count", "gender")]
X = cbind(1, X)
X$gender = ifelse(X$gender == "male", 1, 0)
X = as.matrix(X)
y = reviews$rating_good
```

#### 5.3.4.2 Python

```
import pandas as pd

reviews = pd.read_csv("data/movie_reviews_processed.csv")
X = reviews[['word_count', 'gender']]
```

```
y = reviews["rating_good"]
```

### 5.3.5 Standard Functions

To get started with our first logistic regression model, let's use the `glm` function from R and Python's `statsmodels` function.

#### 5.3.5.1 R

```
model_logistic = glm(
  rating_good ~ word_count + gender,
  data = reviews,
  family = binomial
)

summary(model_logistic)

Call:
glm(formula = rating_good ~ word_count + gender, family = binomial,
     data = reviews)

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  1.71240   0.18136   9.442   <2e-16 ***
word_count -0.14639   0.01551  -9.436   <2e-16 ***
gendermale   0.11891   0.13751   0.865    0.387
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 1370.4 on 999 degrees of freedom
Residual deviance: 1257.4 on 997 degrees of freedom
AIC: 1263.4
```

Number of Fisher Scoring iterations: 4

#### 5.3.5.2 Python

```
import statsmodels.api as sm

X = sm.add_constant(X)

X = pd.get_dummies(X, drop_first = True)

model_logistic = sm.Logit(y, X.astype(float)).fit()
```

```

Optimization terminated successfully.
    Current function value: 0.628697
    Iterations 5

model_logistic.summary()

<class 'statsmodels.iolib.summary.Summary'>
"""

        Logit Regression Results
=====
Dep. Variable: rating_good   No. Observations: 1000
Model: Logit             Df Residuals: 997
Method: MLE               Df Model: 2
Date: Thu, 11 Jan 2024   Pseudo R-squ.: 0.08245
Time: 20:09:34           Log-Likelihood: -628.70
converged: True            LL-Null: -685.19
Covariance Type: nonrobust LLR p-value: 2.925e-25
=====
              coef      std err      z      P>|z|      [0.025      0.975]
-----
const      1.7124     0.181     9.442     0.000      1.357      2.068
word_count -0.1464     0.016    -9.436     0.000     -0.177     -0.116
gender_male 0.1189     0.138     0.865     0.387     -0.151      0.388
=====
"""

```

### 5.3.6 Interpretation and Visualization

We need to know what those results mean. The coefficients that we get from our model are in log odds. We can exponentiate them to get the odds ratio, but we can also exponentiate them and divide by  $1 + \text{that value}$  to get the probability. Interpreting log odds is a fool's errand, but we can at least get a feeling for them directionally. A log odds of 0 would indicate no relationship between the feature and target. A positive log odds would indicate that an increase in the feature will increase the log odds of moving from "bad" to "good", whereas a negative log odds would indicate that a decrease in the feature will decrease the log odds of moving from "bad" to "good". We can convert those log odds to help make some more sense from them.

When we exponentiate the log odds coefficients, we are given the odds ratio. This is the ratio of the odds of the outcome (i.e., success from our binomial distribution) occurring for a one unit increase in the predictor.

```
(Intercept) word_count gendermale
5.5422663  0.8638229  1.1262687
```

Fortunately, the intercept is easy – it is the odds of a "good" review when

word count is 0 and gender is “female”. We see that we’ve got an odds ratio of .86 for the word\_count variable and 1.12 for the male variable. An odds ratio of 1 means that there is no change in the odds of the outcome occurring – essentially that the predictor does not influence the target. An odds ratio of less than 1 means that the odds of the outcome occurring decrease as the predictor increases (while a bit more complicated to wrap your head around, it captures the idea of the odds of moving from a “bad” review to a “good” review decreasing). An odds ratio of greater than 1 means that the odds of the outcome occurring increase as the predictor increases (again, the odds of moving from a “bad” review to a “good” review increasing).

It is far more intuitive to interpret the probability. We can do this by exponentiating the coefficients and dividing by 1 + that value. This will give us the probability of the outcome occurring for a one unit increase in the predictor.

```
(Intercept) word_count gendermale
0.8471478 0.4634683 0.5296925
```

We would say that our probability of moving from a “bad” review to a “good” review is .84 when there are 0 words in the review and the gender is female. Since word\_count is below .5, we know that it will have a negative relationship with the probability of moving from “bad” to “good”; being a male reviewer will have a positive relationship with the probability of moving from “bad” to “good”.

And visualizing those probabilities is absolutely the best way to see how the features influence the target:

TODO: Use output to make a better visual, also use `see` over `sjPlot`

In Figure 5.4, we can see a clear negative relationship between the number of words in a review and the probability of being considered a “good” movie. As we get over 20 words, the predicted probability of being a “good” movie is less than .2.

TODO: Use output to make a better visual

In Figure 5.4, we can see a clear negative relationship between the number of words in a review and the probability of being considered a “good” movie. As we get over 20 words, the predicted probability of being a “good” movie is less than .2. It does not appear that `gender` has much of an effect on the probability of being a “good” movie, since the curves are very similar to each other.

There are interesting issues at play here with regard to our predictor coefficients (what can be considered a *relative effect*) and the model’s effect as a whole on the probability (the *absolute effect*). In circumstances where the intercept is very large (essentially promising a success), the relative effect of a coefficient is practically meaningless. Similarly, very negative coefficients render the relative effects useless.

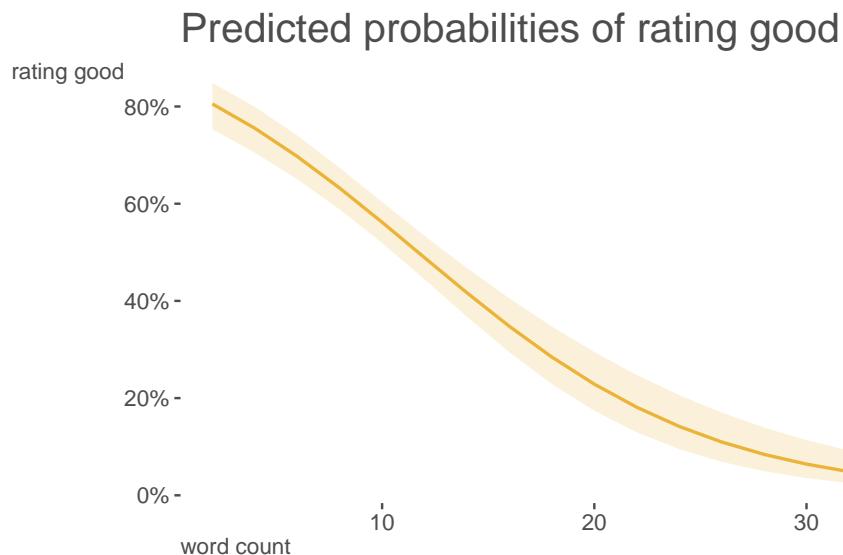


Figure 5.4: Logistic regression predictions for word count feature

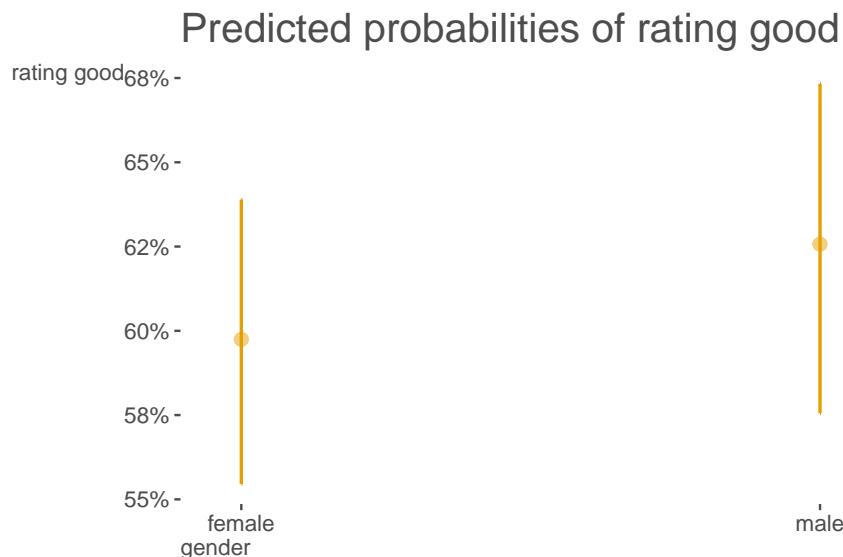


Figure 5.5: Logistic regression predictions for gender feature

### 5.3.7 Objective Function

Let's see how we can pick that work apart to create our own functions. We can use maximum likelihood estimation to estimate the parameters of our model.

#### 5.3.7.1 R

```
logreg_ml = function(par, X, y) {
  beta = par
  N = nrow(X)
  LP = X %*% beta
  mu = plogis(LP)
  L = dbinom(y, size = 1, prob = mu, log = TRUE)
  -sum(L)
}
```

#### 5.3.7.2 Python

```
def logreg_ml(par, X, y):
    beta = par
    N = X.shape[0]
    LP = X.dot(beta).to_numpy()
    mu = [1 / (1 + np.exp(-x)) for x in LP]
    mu_minus_1 = [1 - x for x in mu]
    L = y*np.log(mu) + (1 - y)*np.log(mu_minus_1)
    return -np.sum(L)
```

### 5.3.8 Model Fitting

Now that we have our objective function, we can fit our model. We will use the `optim` function in R and the `minimize` function in Python.

#### 5.3.8.1 R

```
init = rep(0, ncol(X))

names(init) = c('intercept', 'b1', 'b2')

fit_ml = optim(
  par = init,
  fn = logreg_ml,
  X = X,
  y = y,
  control = list(reltol = 1e-8)
)
```

```
pars_ml = fit_ml$par

pars_ml

intercept      b1      b2
1.7121816 -0.1463750 0.1189308
```

### 5.3.8.2 Python

```
import numpy as np
from scipy.optimize import minimize

init = np.zeros(X.shape[1])

fit_ml = minimize(
    fun = logreg_ml,
    x0 = init,
    args = (X, y),
    method = 'BFGS',
    options = {'disp': True}
)

Optimization terminated successfully.
    Current function value: 628.696593
    Iterations: 11
    Function evaluations: 68
    Gradient evaluations: 17

fit_ml.x

array([ 1.71240414, -0.14638763,  0.11891015])
```

! In theory, there is no such thing as 0 or 1 probability. When your model encounters such a value, you may receive a warning, but not an error. The most likely cause of this warning is **separation**: a variable is perfectly separating the target. In other words, once a feature gets below/above a certain value, the target is always 0/1. This can often be caused by very extreme feature values, interaction groups with very small sample sizes, or even accidentally including a function of your target as a feature. More evidence of separation comes when you see your log odds coefficients return something comically large.

**⚠** Logistic regression does not have an  $R^2$  value in the way that a linear regression model does. Instead, there are pseudo- $R^2$  values, but they are not the same as the  $R^2$  value that you are used to seeing. Here is a great breakdown of different pseudo methods.

---

## 5.4 Poisson Regression

### 5.4.1 Why Should You Care

Like logistic regression, poisson regression belongs to a broad class of generalized linear models. Poisson regression is used when you have a count variable as your target. The nature of a count variable is very different, since it starts at 0 and can only be a whole number. We need a model that will not produce negative predictions and poisson regression will do that for us.

### 5.4.2 The Poisson Distribution

The Poisson distribution is very similar to the binomial distribution, but has some key differences. The biggest difference is in its parameter: Poisson has a single parameter noted as  $\lambda$ . This rate parameter is going to estimate the expected number of events during a time interval. This can be accidents in a year, pieces produced in a day, or hits during the course of a baseball season. We can find the rate by determining the number of events per interval, multiplied by the interval length.

$$\frac{\text{event}}{\text{interval}} * \text{interval length}$$

To put some numbers to that, if we have 1 accident per week in a factory and we are observing a whole year, we would have a rate of  $(1/7)*28 = 4$  accidents per month.

Let's see what that particular distribution might look like in Figure 5.6:

We can also see what it looks like for different rates (some places might be safer than others) in Figure 5.7:

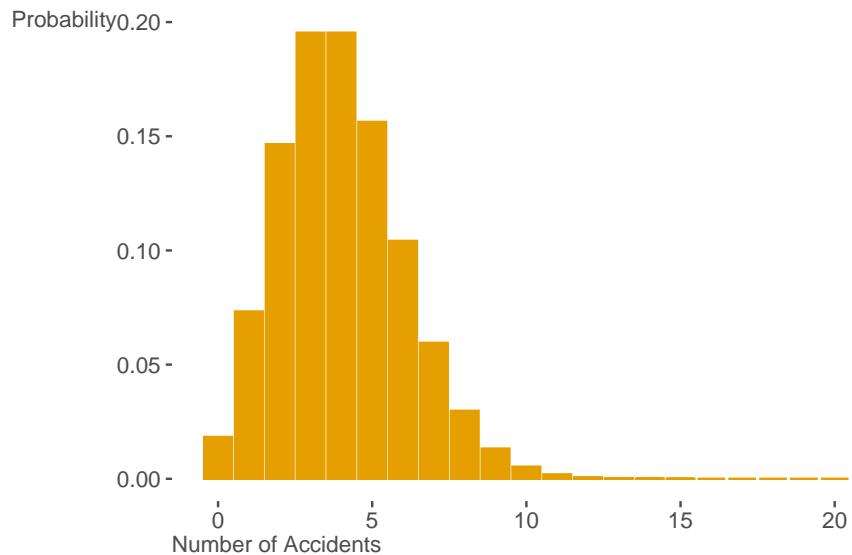


Figure 5.6: Poisson distribution for a rate of 4

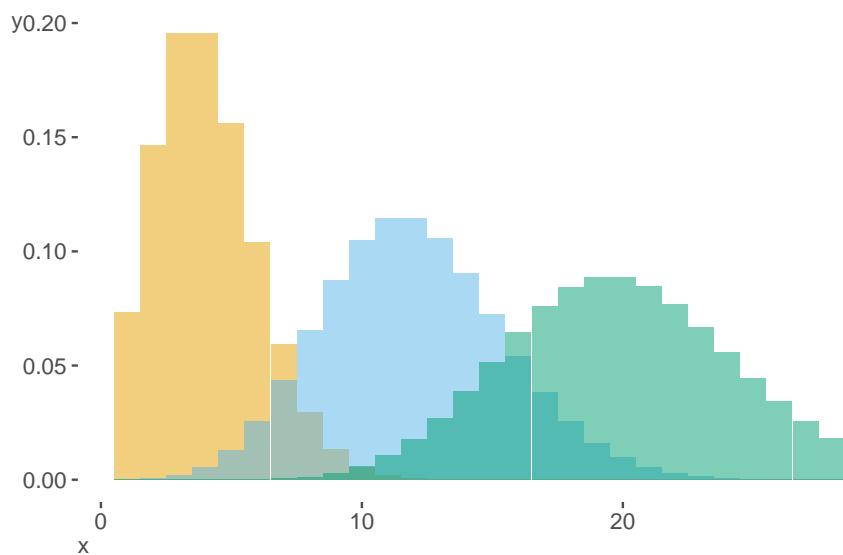


Figure 5.7: Poisson distributions for different rates

**i** A cool thing about these distributions is that they can deal with different *exposure* rates. You don't need observations recorded over the same interval length, because you can adjust for them appropriately. They can also be used to model inter-arrival times and time-until events.

Let's make a new variable that will count the number of times a person uses a personal pronoun word.

#### 5.4.2.1 R

```
reviews$poss_pronoun = stringr::str_count(
  reviews$review_text,
  "\bI\b|\bme\b|\b[Mr]y\b|\bmine\b|\bmyself\b"
)
```

#### 5.4.2.2 Python

```
reviews['poss_pronoun'] = reviews['review_text'].str.count(
  "\bI\b|\bme\b|\b[Mr]y\b|\bmine\b|\bmyself\b"
)
```

#### 5.4.3 The (Sometimes) Thin Line

Let's think long and hard about our target variable and what it actually might be. Since Poisson regression gets its name from the Poisson distribution, we should probably see if it follows the Poisson distribution.

```
Goodness-of-fit test for poisson distribution

X^2 df P(> X^2)
Likelihood Ratio 2.283728 3 0.5156453
```

This is a  $\chi^2$  to test if the distribution deviates from a Poisson. If we see a statistically significant value, we would say that it deviates from the tested distribution. In this case, it is pretty clear that `poss_pronoun` could come from a Poisson distribution.

We can also plot that test using a hanging rootogram:

TODO: convert to ggplot

In Figure 5.8, the bars are the observed counts and the red line/points are the fitted counts (i.e., how many would be expected). If a bar does not reach the 0 line, then the model would over-predict for that particular count; if the bar dips below the 0 line, the model under-predicts that count. It looks like we are pretty close for our counts.

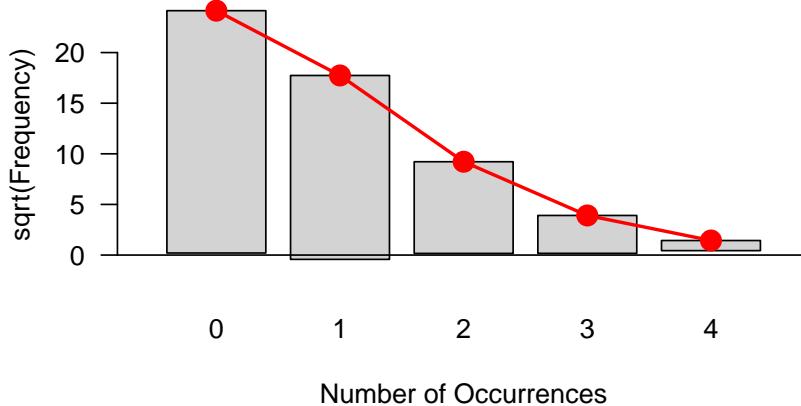


Figure 5.8: Hanging rootogram for Poisson distribution

#### 5.4.4 Standard Functions

Recall that every distribution has a link function (or several) that tend to work well for it. The poisson distribution uses a log link function:

$$\begin{aligned} \log(\lambda) &= \alpha + X\beta \\ y &= \text{Poisson}(\lambda) \end{aligned}$$

Using the log link keeps the outcome positive (we cannot deal with negative counts). Logs, as they are prone to do, are going to tend towards an exponential relationship; just be sure that it makes sense over the entire range of your data.

##### 5.4.4.1 R

```
model_poisson = glm(
  poss_pronoun ~ word_count,
  data = reviews,
  family = poisson
)

summary(model_poisson)

Call:
```

```

glm(formula = poss_pronoun ~ word_count, family = poisson, data = reviews)

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -1.848982  0.099409 -18.60 <2e-16 ***
word_count   0.103126  0.006433  16.03 <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

Null deviance: 996.21 on 999 degrees of freedom
Residual deviance: 776.19 on 998 degrees of freedom
AIC: 1699.7

Number of Fisher Scoring iterations: 5

exp(model_poisson$coefficients)

(Intercept) word_count
0.1573974  1.1086314

```

#### 5.4.4.2 Python

```

import statsmodels.api as sm
import statsmodels.formula.api as smf

model_poisson = smf.glm(
    formula = "poss_pronoun ~ word_count",
    data = reviews,
    family = sm.families.Poisson()
).fit()

model_poisson.summary()

<class 'statsmodels.iolib.summary.Summary'>
"""
                                         Generalized Linear Model Regression Results
=====
Dep. Variable:      poss_pronoun    No. Observations:          1000
Model:                 GLM        Df Residuals:                  998
Model Family:           Poisson     Df Model:                      1
Link Function:            Log        Scale:                   1.0000
Method:                IRLS      Log-Likelihood:             -847.83
Date:      Thu, 11 Jan 2024    Deviance:                  776.19
Time:      20:09:35      Pearson chi2:                     717.
No. Iterations:             5    Pseudo R-squ. (CS):            0.1975

```

```
Covariance Type: nonrobust
=====
            coef    std err      z   P>|z|      [0.025     0.975]
-----
Intercept   -1.8490    0.099  -18.599    0.000    -2.044    -1.654
word_count   0.1031    0.006   16.030    0.000     0.091     0.116
=====
"""
np.exp(model_poisson.params)

Intercept    0.157397
word_count   1.108631
dtype: float64
```

We are going to interpret this almost the same as a linear regression. The slight wrinkle here, though, is that we are looking at the log counts (remember that we specified the log link function). In other words, an increase in one review word leads to an expected log count increase of ~.01. Just like our logistic regression, we could exponentiate this to get 1.108 – every added word in a review gets us a ~1% increase in the number of possessive pronouns. Let's see what this looks like in action in Figure 5.9:

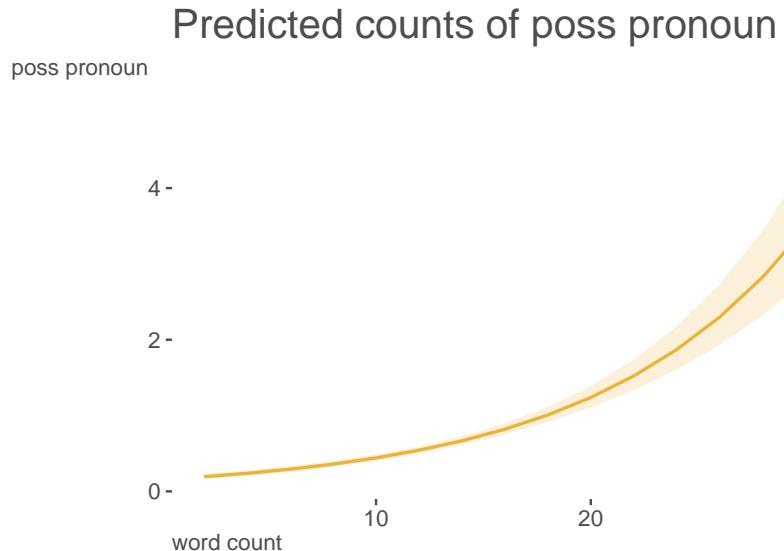


Figure 5.9: Poisson regression predictions for word count feature

With everything coupled together, we have a meaningful coefficient for `word_count`, a clear plot, and adequate model fit. Therefore, we might conclude

that there is a positive relationship between number of words in a review on the number of times a person uses a personal possessive.

```
Overdispersion test
```

```
data: model_poisson
z = -8.0493, p-value = 1
alternative hypothesis: true dispersion is greater than 1
sample estimates:
dispersion
0.7606014
```

The dispersion value that we see returned (0.7606014 in our case) should be under 1. A dispersion value over 1 means that we have overdispersion. Our dispersion value, coupled with our high  $p$ -value, indicates that we would fail to reject the null hypothesis of equidispersion.

We can also look back to our model results to compare our residual deviance to our residual deviance degrees of freedom; if our deviance is greater than our degrees of freedom, we might have an issue with overdispersion. Since we are just a bit over and our overdispersion tests do not indicate any huge issue, we can be relatively okay with our model. If we had some more extreme overdispersion, we would want to flip to a quasi-poisson distribution – our coefficients would not change, but we would have improved standard errors.

### 5.4.5 Model Specification

TODO: Need some text here

#### 5.4.5.1 R

```
pois_ll = function(y, X, par) {
  beta = par
  lambda = exp(beta%*%t(X))
  loglik = -sum(dpois(y, lambda, log = TRUE))
  return(loglik)
}
```

#### 5.4.5.2 Python

```
from scipy.stats import poisson

def pois_ll(par, X, y):
    beta = par
    lambda_ = np.exp(X.dot(beta))
    loglik = -np.sum(poisson.logpmf(y, lambda_))
    return loglik
```

### 5.4.6 Model Fitting

#### 5.4.6.1 R

```
form = as.formula("poss_pronoun ~ word_count")
model = model.frame(form, data = reviews)
X = model.matrix(form, data = reviews)
y = model.response(model)

starts = c(0, 0)

fit = optim(
  par = starts ,
  fn  = pois_ll,
  X   = X,
  y   = y,
  method = "BFGS",
  hessian = TRUE
)

fit$par
```

[1] -1.8487431 0.1031103

#### 5.4.6.2 Python

```
X = np.column_stack((np.ones(reviews.shape[0]), reviews[['word_count']]))

y = reviews["poss_pronoun"]

init = np.zeros(X.shape[1])

fit = minimize(
  fun = pois_ll,
  x0 = init,
  args = (X, y),
  method = 'BFGS'
)
```

fit.x

array([-1.84898106, 0.10312625])

## 5.5 Wrapping Up

These are just two of the many models that fall under the broad umbrella of generalized linear models. Depending on your data situation, you might want to keep Table 5.1 in mind:

Table 5.1: Targets and distributions for generalized linear models

Target	Distribution
Proportions	binomial/beta
Exponential response	gamma
3+ categories	multinomial
Count	poisson/negative binomial

That is, however, just a tiny slice of the potential distributions that you might find yourself needing to use in a similar way. While not all are considered official ‘generalized linear models’, the approach is the same. While you could always use the general linear model, the key is to understand the distribution of your target and then find the appropriate link function to connect it to the linear model. Using the proper distribution will yield better results and get your model a little closer to the answer you seek.

---

## 5.6 Additional Resources

In any given graduate coursework, you might find a whole semester dedicated to GLMs. We’ve only scratched the surface here, but there are some great resources out there to help you dig deeper. If you are itching for a text book, there isn’t any shortage of them out there and you can essentially take your pick. If you are looking for something a bit more applied to get you going, you might want to check out Roback and Legler’s *Beyond Multiple Linear Regression*, available for free at <https://bookdown.org/robback/bookdown-BeyondMLR/>.



# 6

---

## *Extending the Linear Model*

---

With thin linear model and generalized linear models, we have a solid foundation for modeling. We've seen how there is a notable amount we can do with a conceptually simple approach. We've also seen how we can extend the linear model to handle different types of data to help us understand and make some inferences about the relationships between our features and target.

In this chapter, we want to show you how to extend the linear model even further with still common tools. These particular methods are also good examples of how we can think about our data and approach in different ways, and can serve as a good starting point for even more techniques you may want to explore in the future.

---

### 6.1 Key Ideas

- The linear and generalized linear models are great and powerful starting points for modeling, but they are not the only options. But it's good to know a few more ways we can extend the approach.
- Quantile regression, GAMs, and mixed models allow us to model relationships that are not linear or monotonic, and can help us to better understand our data.
- While these seem like very different approaches, we can still use our linear model concepts and approach at the core, take similar estimation steps, and even have similar, albeit more, interpretation.

#### 6.1.1 Why this matters?

The linear model is a great starting point for modeling. It is a simple approach that can be used to model a wide variety of relationships between features and targets. It is also a great way to get a feel for how to think about modeling. But linear and generalized models are just the starting point, and the models depicted here are very common extensions used in a variety of disciplines and industries.

### 6.1.2 Good to know

While these models are extensions of the linear model, they are not necessarily more complex. They are also not necessarily more difficult to interpret. However, you likely want to be fairly comfortable with standard linear models at least before you start to explore these extensions.

---

## 6.2 Quantile Regression

Oh, you think the mean is your ally. But you merely adopted the mean; I was born in it, molded by it. I didn't see anything interesting until I was already a man. And by then, it was nothing to me but illuminating. – Bane (probably)

People generally understand the concept of the arithmetic mean. You see it some time during elementary school, it gets tossed around in daily language (usually using the word “average”), and it is statistically important. After all, where would the normal distribution be without a mean? Why, though, do we feel so tied to it from a regression modeling perspective? Yes, it has handy features, but it is also a bit restrictive to the types of relationships that it can actually model well.

In this chapter, we want to show you what to do when the mean betrays you – and trust us, the mean will betray you at some point!

### 6.2.1 Why Should You Care?

Sometimes the mean doesn’t make as much sense to focus on for summarizing our data, whether due to extreme scores that have too much influence, or you’re interested at how your model is performing in different parts of your data, such as the top 10%, quantile regression can be helpful. Quantile regression will give you the ability to model the relationship between your features and target at different quantiles of your target, with the median being a starting point. Maybe people who are older are more likely to rate movies higher, but that relationship is stronger for people who rate movies higher than the median. Quantile regression will let you model that relationship.

### 6.2.2 When The Mean Breaks Down

In a perfect data world, we like to assume the mean is equal to the middle observation of the data: the *median*. But that is only when things are symmetric though, and usually our data comes loaded with challenges. Skewness

and even just a few extreme scores in your data may cause a rift between the median and the mean.

Let's say we take the integers between 1 and 10, and find the mean.

$$\frac{1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10}{10} = 5.5$$

The middle value in that vector of numbers would also be 5.5.

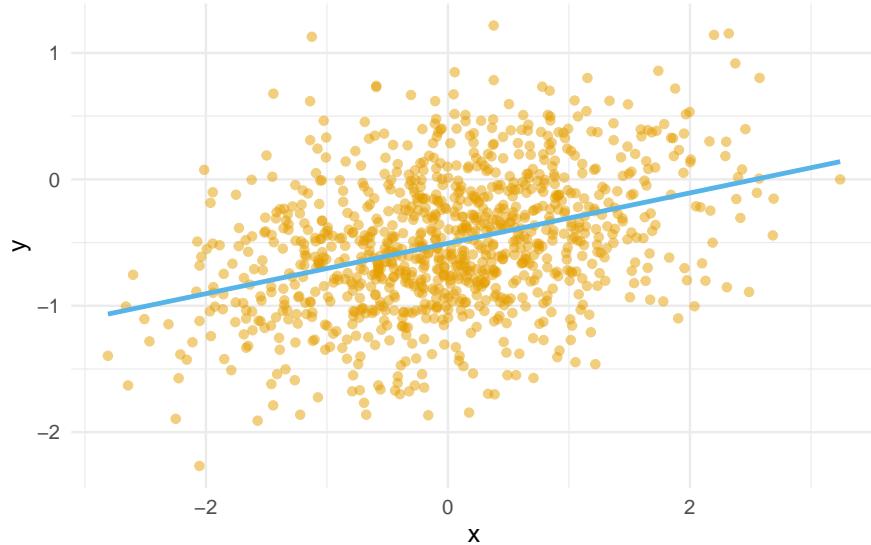
What happens we replace the 1 with a more extreme value, like -10?

$$\frac{-10 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10}{10} = 4.5$$

With just one dramatic change, our mean went down by a whole point. The median observation, though, is still 5.5. In short, the median is invariant to wild swings out in the tails of your numbers.

You might be saying to yourself, “Why should I care about this central tendency chicanery?” Let us tell you why you should care – the least squares approach to the standard linear model dictates that the regression line needs to be fit through the means of the variables. If you have extreme scores that influence the mean, then your regression line will also be influenced by those extreme scores.

Let's look at a few different regression lines:



Now, what would happen if we replaced a few of our observations with extreme scores?

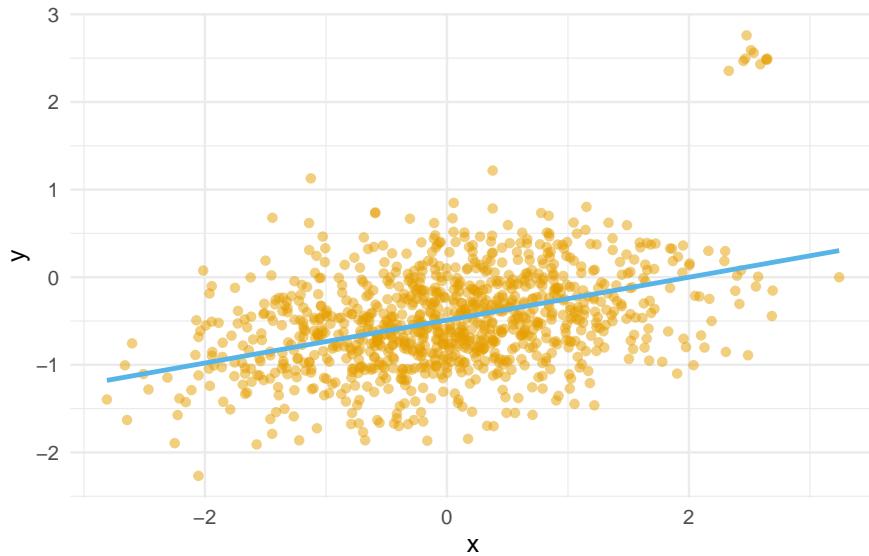


Figure 6.1: Linear line with extreme scores

With just a casual glance, it doesn't look like our two regression lines are that different. They both look like they have a similar positive slope, so all should be good. To offer a bit more clarity, though, let's put those lines in the same space:

With 1000 observations, we see that having just 10 extreme scores is enough to change the regression line, even if just a little. But that little bit can mean a huge difference for predictions or just the conclusions we come to.

There are a few approaches we could take here, with common approaches being dropping those observations or Winsorizing them. Throwing away data because you don't like the way it behaves is nearing on statistical abuse and Winsorization is just replacing those extreme values with numbers that you like a little bit better. Let's not do that!

A better answer to this challenge might be to not fit the regression line through the mean, but the median instead. This is where quantile regression becomes handy. Formally, this model can be expressed as:

$$Q_{Y|X}(\tau) = X\beta_\tau$$

Where we can find the estimation of  $\beta_\tau$  as:

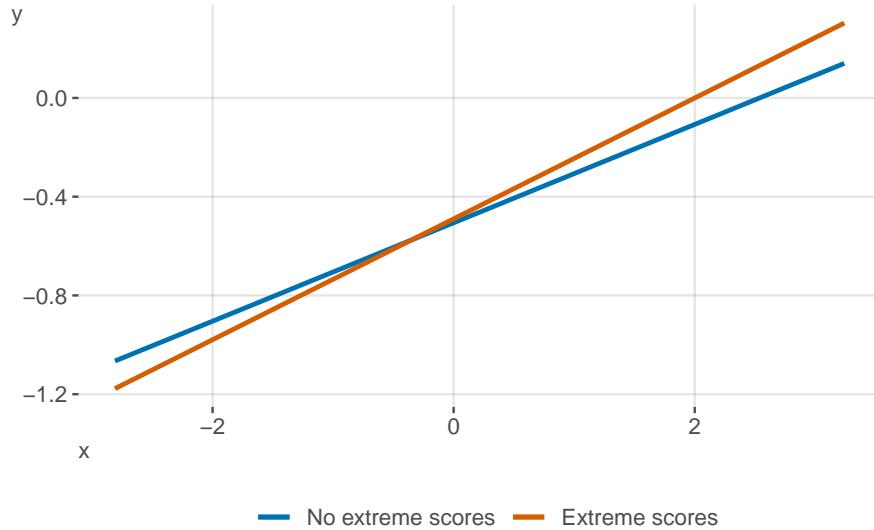


Figure 6.2: Line lines with and without extreme scores

TODO: change a bit to other loss function depiction, add the explicit quantile function

$$\begin{aligned}\hat{\beta}_\tau &= \arg \min_{\beta \in \mathbb{R}^k} \sum_{i=1}^n (\rho_\tau(Y_i - X_i \beta)) \\ &= \arg \min_{q \in \mathbb{R}} \left[ (\tau - 1) \sum_{y_i < q} (y_i - q) + \tau \sum_{y_i \geq q} (y_i - q) \right]\end{aligned}$$

With quantile regression, we are given an extra parameter for the model:  $\tau$  or *tau*. The tau parameter let's us choose which quantile we want to use for our line fitting. Since the median splits the data in half, we can translate that to a quantile of .5.

### 6.2.3 Data Import and Preparation

Let's bring in our movie reviews data. Let's say that we are curious about the relationship between the `total_reviews` variable and the `rating` variable. First, we need to get our data ready for our home-brewed functions.

TODO: Rescale total\_reviews or use the scaled version

### 6.2.3.1 R

```
reviews = read.csv("data/movie_reviews_processed.csv")

reviews = na.omit(reviews)

X = reviews$total_reviews
X = cbind(1, X)
y = reviews$rating
```

### 6.2.3.2 Python

```
import pandas as pd
import numpy as np

reviews = pd.read_csv("data/movie_reviews_processed.csv")

reviews = reviews.dropna()

X = pd.DataFrame(
    {'intercept': 1,
     'total_reviews': reviews['total_reviews']}
)
y = reviews['rating']
```

### 6.2.4 Standard Functions

We can see how we can use `quantreg` and `statsmodels` to create a quantile regression. For both, we will start with a median regression; in other words, a quantile of .5.

#### 6.2.4.1 R

```
library(quantreg)

median_test = rq(rating ~ total_reviews, tau = .5,
                 data = reviews)

summary(median_test)

Call: rq(formula = rating ~ total_reviews, tau = 0.5, data = reviews)

tau: [1] 0.5

Coefficients:
            coefficients lower bd upper bd
(Intercept) 2.70278     2.53409  2.77655
```

```
total_reviews 0.00007      0.00006  0.00010
```

#### 6.2.4.2 Python

```
import statsmodels.formula.api as smf

median_test = smf.quantreg('rating ~ total_reviews',
                           data = reviews).fit(q = .5)

median_test.summary()

<class 'statsmodels.iolib.summary.Summary'>
"""

QuantReg Regression Results
=====
Dep. Variable: rating Pseudo R-squared: 0.05241
Model: QuantReg Bandwidth: 0.3092
Method: Least Squares Sparsity: 1.645
Date: Sun, 17 Dec 2023 No. Observations: 1000
Time: 20:07:49 Df Residuals: 998
Df Model: 1
=====
coef std err t P>|t| [0.025 0.975]
-----
Intercept 2.6929 0.052 51.706 0.000 2.591 2.795
total_reviews 7.361e-05 9.17e-06 8.029 0.000 5.56e-05 9.16e-05
=====
```

The condition number is large, 1.14e+04. This might indicate that there are strong multicollinearity or other numerical problems.

""

Fortunately, our interpretation of this result isn't all that different from a standard linear model – the rating should increase by .00007 for every additional review. However, this is at rating median, not the mean, like the standard linear model.

Quantile regression is not a one-trick-pony. Remember, it is called quantile regression – not median regression. Being able to compute a median regression is just the default. What we can do also is to model different quantiles of the same data. It gives us the ability to answer brand new questions – does the relationship between user age and their ratings change at different quantiles of rating? Very cool!

Instead of a single model to capture the trend through the mean of the data, we can now examine the trends within 5 different quantiles of the data (we aren't limited to just those quantiles, though, and you can examine any of

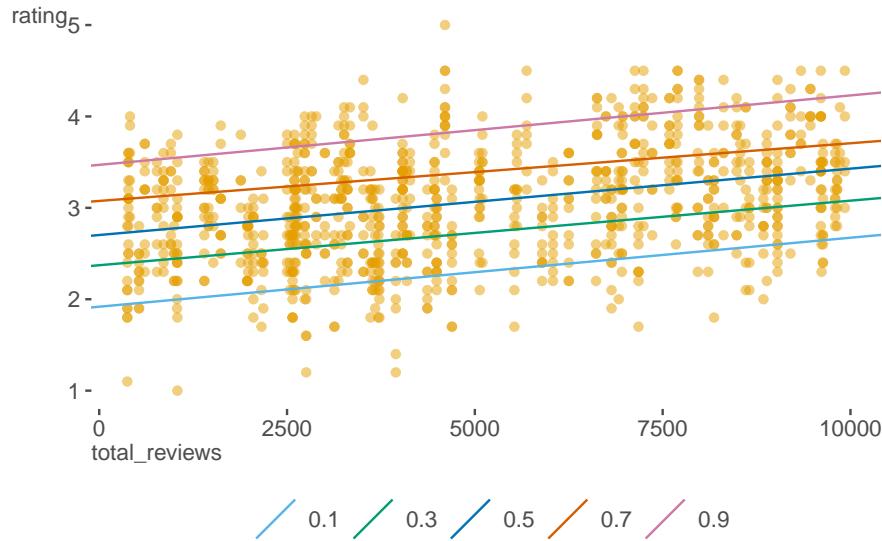


Figure 6.3: Quantile regression lines

them that you might find interesting). If we had to put some words to our visualization, we could say that all of the quantiles show a positive relationship. While they all appear to have roughly the same slope, it appears that the 90th quantile has a slightly steeper slope than the other quantiles, if only modestly so.

TODO: MOVE TO ESTIMATION OR ONLINE ONLY

### 6.2.5 Quantile Loss Function

Now that we know how to use standard functions for quantile regression, let's see one way that we can create a least squares loss function for fitting a linear regression model and compare it with a function for quantile loss.

#### 6.2.5.1 R

```
least_squares_loss = function(par, X, y) {
  linear_parameters = X %*% par
  mu = linear_parameters
  loss = crossprod(y - mu)
}
```

```
quantile_loss = function(par, X, y, tau) {

  linear_parameters = X %*% par

  residual = y - linear_parameters

  loss = ifelse(
    residual < 0 ,
    (-1 + tau)*residual,
    tau*residual
  )

  sum(loss)
}
```

### 6.2.5.2 Python

```
def least_squares_loss(par, X, y):
  linear_parameters = X.dot(par)

  mu = linear_parameters

  loss = np.dot(y - mu, y - mu)

  return loss

def quantile_loss(par, X, y, tau):
  linear_parameters = X.dot(par)

  residual = y - linear_parameters

  loss = []

  for i in residual:
    if i < 0: loss.append((-1 + tau)*i)
    else: loss.append(tau*i)

  return sum(loss)
```

You'll notice right away that we have a few differences. Our quantile loss function includes the **tau** argument, which will let us set our quantile of interest; naturally, it can be any value between 0 and 1. The residual is multiplied by the tau value, only if the residual is greater than 0. If the residual is negative, we need to add tau to -1. Since we need a positive value for our loss values, we will multiply our negative residuals by the negative value produced from

-1 plus our tau value. After that, we just sum all of those positive loss values and do our best to minimize that summed value.

### 6.2.6 Model Fitting

Now that we have our data and our loss function, we can fit the model almost exactly like our standard linear model. Again, note the difference here with our tau value, which we've set to .5 to represent the median.

#### 6.2.6.1 R

```
optim(
  par = c(intercept = 0, total_reviews = 0),
  fn  = quantile_loss,
  X   = X,
  y   = y,
  tau = .5
)$par

  intercept total_reviews
2.702730e+00 7.258694e-05
```

#### 6.2.6.2 Python

```
from scipy.optimize import minimize

minimize(
  quantile_loss,
  x0 = np.array([0, 0]),
  args = (X, y, .5)
).x

array([2.70270449e+00, 7.26658864e-05])
```

---

## 6.3 Additive Models

Wiggle, wiggle, wiggle, yeah! – LMFAO

TODO: GAM gets a bit deep, but we do want to keep discussion of penalized approach, and ultimately a word on the random effect connection.

### 6.3.1 Why Should You Care?

Not every relationship is linear and not every relationship is monotonic. Sometimes, you need to be able to model a relationship that has a fair amount of nonlinearity – they can appear as slight curves, waves, and any other type of wiggle that you can imagine. Additive models will give you the ability to model those nonlinear relationships between your features and target.

### 6.3.2 When Straight Lines Aren't Enough

Fitting a line through your data is always going to be useful, regardless of whether you are using the median or the mean. Those lines give us a wonderful ability to say important things about the relationships between variables and how one variable might influence another. What if we just want to dispense with the notion that we need to fit a straight line through some mass of the data? What if we relax the idea that we need a straight line and think in terms of fitting something curvy through the data.

In other words, we can go from the straight line in Figure 6.4:

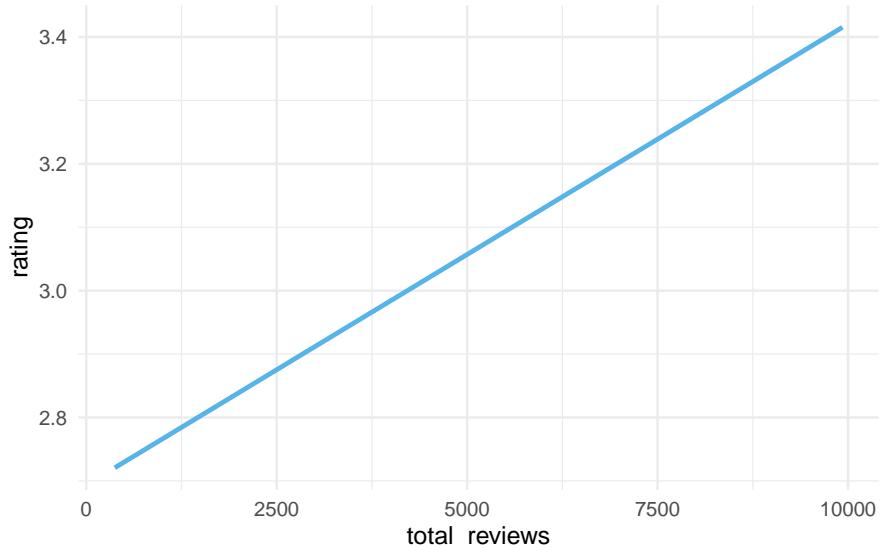


Figure 6.4: A standard linear model

TODO: there is actually a poly effect for length in minutes, but it may not

To the curve seen here:

That curved line in Figure 6.5 is called a spline. Oddly enough, we can still use a linear model to fit this spline through the data. While this might not give

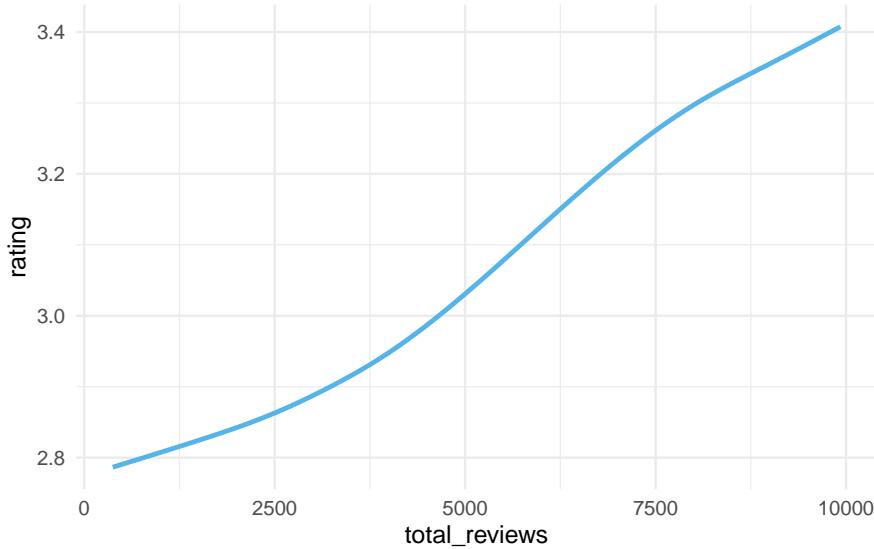


Figure 6.5: A generalized additive model

us the same tidy explanation that a typical line would offer, we will certainly get better prediction.

These models belong to a broad group of *generalized additive models*, often shortened to GAMs. When we fit a quantile regression, we made a slight tweak to the  $y$ ; to fit a GAM, we are going to tweak our predictors. How are we going to tweak them, you might ask? We are essentially going to let them be an *additive* function of features. We will have a model that looks like this:

$$y = f(x) + \epsilon$$

This isn't much different than before, and technically, it really isn't. It's the same linear combination of features we have with a basic linear model. The difference is that we are going to let  $f(x)$  be a function of  $x$  that allows us to capture other types of relationships by expanding the feature  $x$  in different ways, some that can be complex, but on the practical side are just extra columns in the **model matrix**. If you are familiar with polynomial regression, you can think of this as a generalization of that approach, but which is identical in spirit<sup>1</sup>.

These additive features will allow us to capture nonlinearities in our data

---

<sup>1</sup>If we penalized the coefficients of the polynomial regression, we would have a penalized spline. It's not used that much in GAMs these days, but there's nothing wrong with doing so. See Section 4.9 for more information on penalized regression.

very nicely. At this point, you might be asking yourself, “Why couldn’t I just use some type of polynomial regression or even a nonlinear regression?”. Of course you could, but both have limitations. The typical polynomial regression tends to overfit the data you currently have, and **you** are forcing curves to fit through the data. To use a nonlinear model, you need to know what the underlying nonlinear form actually looks like before you can even specify the model, and without extra steps, will also tend to overfit. A GAM handles this situation a little better in that it will produce a curve that will best fit through the data, without the need to know the underlying functional form.

### 6.3.3 Standard Functions

Now that you have some background, let’s . In most respects, we can use the same sort of approach as we did with our other model examples.

#### 6.3.3.1 R

We’ll use the very powerful `mgcv` package in R. The `s` function will allow us to use a spline approach to capture the nonlinearity.

```
library(mgcv)

gam_model = gam(rating ~ s(total_reviews_sc, bs = "cr"), data = reviews)

summary(gam_model)

Family: gaussian
Link function: identity

Formula:
rating ~ s(total_reviews_sc, bs = "cr")

Parametric coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) 3.05140   0.01862   163.9 <2e-16 ***
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Approximate significance of smooth terms:
          edf Ref.df    F p-value
s(total_reviews_sc) 8.672 8.966 16.2 <2e-16 ***
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

R-sq.(adj) = 0.124  Deviance explained = 13.1%
GCV = 0.34994  Scale est. = 0.34655 n = 1000
```

### 6.3.3.2 Python

We use the `pygam` package in Python to fit a GAM. It's very much in the spirit of R's `mgcv`. The `s` function will allow us to use a spline approach to capture the nonlinearity.

```
from pygam import LinearGAM, s

y = reviews['rating']

X = pd.DataFrame({'total_reviews_sc': reviews['total_reviews_sc']})

gam_model = LinearGAM(s(0, n_splines=10)).fit(X, y)

gam_model.summary()

LinearGAM
=====
Distribution: NormalDist Effective DoF: 7.3297
Link Function: IdentityLink Log Likelihood: -1286.4909
Number of Samples: 1000 AIC: 2589.6412
AICc: 2589.7981
GCV: 0.3553
Scale: 0.3506
Pseudo R-Squared: 0.119
=====
Feature Function Lambda Rank EDoF P > x Sig. Code
=====
s(0) [0.6] 10 7.3 1.11e-16 ***
intercept 1 0.0 1.11e-16 ***
=====
Significance codes: 0 *** 0.001 ** 0.01 * 0.05 . 0.1 ' ' 1

WARNING: Fitting splines and a linear function to a feature introduces a model identifiability problem
which can cause p-values to appear significant when they are not.

WARNING: p-values calculated in this manner behave correctly for un-penalized models or models with
known smoothing parameters, but when smoothing parameters have been estimated, the p-values
are typically lower than they should be, meaning that the tests reject the null too readily.
```

For the results of our gam model, one of the best places to look first is the `edf/EDoF` column. This column indicates the *effective degrees of freedom*. You can think of it as a measure of wiggle in the relationship between the predictor and the target. The higher the value, the more wiggle you have. If you have a value close to 1, then you have a linear relationship. With our current result, we can be pretty confident that a nonlinear relationship gives a better idea about the relationship between `total_reviews` and `rating` than a linear relationship.

Depending on the tool we used, we may also get information about our Adjusted  $R^2$  (**R-sq.(adj)**) and **Deviance explained**, which is an analog to the unadjusted  $R^2$  value for a Gaussian model. We also have **GCV** – the generalized cross validation score. It is an estimate of the mean square prediction error based on a leave-one-out cross validation estimation process. Naturally, the lower the GCV, the better the model.

TODO: NEED VISUAL

Beyond this the real approach to interpretation is in the visualization. We can use the `plot` function to get a sense of the relationship between our features and target.

### 6.3.3.3 R

```
plot(gam_model, se = TRUE)
```

### 6.3.3.4 Python

```
from pygam.utils import generate_X_grid
import matplotlib.pyplot as plt

X_grid = generate_X_grid(gam_model)

plt.plot(X_grid, gam_model.predict(X_grid), color = "#00AAFF")
plt.plot(X_grid, gam_model.prediction_intervals(X_grid), color = "#00AAFF", ls = '--')
plt.scatter(X, y, alpha = .5)
```

Unfortunately the default package plots are not pretty, and sadly aren't provided in the same way we'd expect for interpretation. But they're fine for looking at your wiggly result. But we provide a better looking one<sup>2</sup>. The main interpretation is that we're mostly flat for total reviews less than the mean. At that point we get more of a positive linear effect, followed by a bit of a mixed bag for movies with lots of total reviews.

TODO: Move to estimation, but may need to nix for pdf brevity, and needs some work to fit well.

## 6.3.4 Splines

Now that you've gotten a taste of a standard way of specifying a GAM, let's roll our own. We are going to need to generate several functions to make this work. The first will be to produce the *cubic spline*. Do take note that there are many different types of splines that could be used.

---

<sup>2</sup>We used the `see` in R for a quick plot. We also recommend its functionality via the `gratia` package to visualize the derivatives, which will show more of where the effect is changing most.

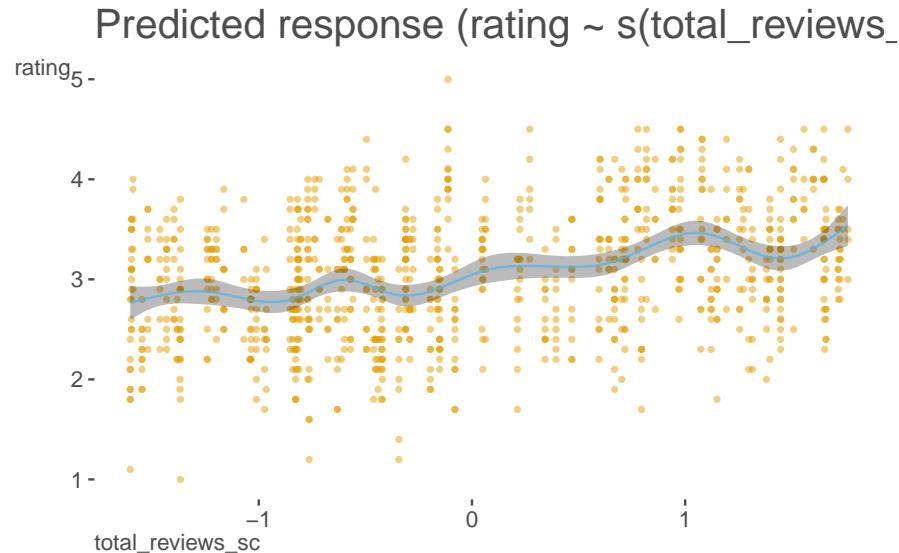


Figure 6.6: Visualizing a GAM

#### 6.3.4.1 R

```
cubic_spline = function(x, z) {
  ((z - 0.5)^2 - 1/12) * ((x - 0.5)^2 - 1/12)/4 -
  ((abs(x - z) - 0.5)^4 - (abs(x - z) - 0.5)^2 / 2 + 7/240) / 24
}
```

#### 6.3.4.2 Python

```
import numpy as np

def cubic_spline(x,z):
    return (((z - 0.5)**2 - 1/12) * ((x - 0.5)**2 - 1/12)/4 -
           ((np.abs(x - z) - 0.5)**4 - (np.abs(x - z) - 0.5)**2 / 2 + 7/240) / 24)
```

#### 6.3.5 Model Matrix Function

Then we a function to produce the model matrix:

##### 6.3.5.1 R

```
splX = function(x, knots) {
  q = length(knots) + 2      # number of parameters
  n = length(x)              # number of observations
  X = matrix(1, n, q)        # initialized model matrix
```

```
X[, 2] = x                      # set second column to x
X[, 3:q] = outer(x, knots, FUN = cubic_spline)
x
}
```

### 6.3.5.2 Python

```
def splX(x, knots):
    q = len(knots) + 2
    n = len(x)
    X = np.ones((n, q))
    X[:, 1] = x
    for i in range(2, q):
        X[:, i] = cubic_spline(x, knots[i-2])
    return X
```

This model matrix will help us to produce an *unpenalized spline*.

### 6.3.6 Model Matrix

We can create a model with 4 knots – you can think of knots as places where individual regression lines will get joined together. You can always experiment with more or less knots. Once we have our knots ready, we can create the model matrix.

As soon as you create your `x` object, you should take a look at it. It will be a matrix with 4 columns. The first column will be all 1s, the second column will be the scaled `user_age`, and the last two columns will be the cubic splines.

#### 6.3.6.1 R

```
knots = 1:4/5
rating = reviews$rating
x = reviews$total_reviews_sc
X = splX(x, knots)
head(X)

 [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
[1,] 1 -0.8272845 4.153406e-03 -0.0326229161 -0.0448777445 -0.0392293550
[2,] 1  0.9402239 3.231161e-05 -0.0032068475 -0.0027357582  0.0009673706
[3,] 1 -1.1641541 -4.416516e-03 -0.0803004886 -0.1235780974 -0.1435625745
[4,] 1  0.4683950 -3.548729e-04  0.0027291926  0.0023541418 -0.0007952384
[5,] 1  0.3940441 2.496222e-04  0.0027093841  0.0015964818 -0.0011544588
[6,] 1  0.7781318 -1.099612e-03 -0.0008054941  0.0006053991  0.0013598650
```

### 6.3.6.2 Python

```

knots = np.arange(1, 5) / 5

x = reviews['total_reviews_sc']

rating = reviews['rating']

X = splX(x, knots)

X[:5, :]

array([[ 1.0000000e+00, -8.27284454e-01,  4.15340591e-03,
       -3.26229161e-02, -4.48777445e-02, -3.92293550e-02],
       [ 1.0000000e+00,  9.40223866e-01,  3.23116121e-05,
       -3.20684754e-03, -2.73575817e-03,  9.67370635e-04],
       [ 1.0000000e+00, -1.16415406e+00, -4.41651564e-03,
       -8.03004886e-02, -1.23578097e-01, -1.43562575e-01],
       [ 1.0000000e+00,  4.68394990e-01, -3.54872863e-04,
       2.72919264e-03,  2.35414179e-03, -7.95238367e-04],
       [ 1.0000000e+00,  3.94044062e-01,  2.49622200e-04,
       2.70938406e-03,  1.59648177e-03, -1.15445884e-03]])

```

### 6.3.7 Model Fitting

Now that we have a model matrix,  $x$ , we can fit the model. All of the hardwork was done in creating the model matrix and we can just use `lm` or `OLS` to fit the model.

#### 6.3.7.1 R

```

fit_lm = lm(rating ~ X - 1)

fit_lm

Call:
lm(formula = rating ~ X - 1)

Coefficients:
            X1          X2          X3          X4          X5          X6
2.9776     0.3493    31.7281   -99.7536   101.4611   -34.2938

```

#### 6.3.7.2 Python

```

import statsmodels.api as sm

fit_lm = sm.OLS(rating, X).fit()

```

```

fit_lm.summary()

<class 'statsmodels.iolib.summary.Summary'>
"""

            OLS Regression Results
=====
Dep. Variable:          rating    R-squared:       0.116
Model:                 OLS     Adj. R-squared:    0.111
Method:                Least Squares   F-statistic:      26.03
Date:        Sun, 17 Dec 2023   Prob (F-statistic):  9.31e-25
Time:        20:07:52           Log-Likelihood:   -893.05
No. Observations:      1000    AIC:             1798.
Df Residuals:          994    BIC:             1828.
Df Model:                  5
Covariance Type:        nonrobust
=====
            coef    std err         t      P>|t|      [0.025      0.975]
-----
const    2.9776    0.050    59.862    0.000     2.880     3.075
x1      0.3493    0.057     6.136    0.000     0.238     0.461
x2     31.7281   13.952     2.274    0.023     4.349    59.107
x3     -99.7536   45.771    -2.179    0.030    -189.573   -9.934
x4     101.4611   45.666     2.222    0.027    11.849   191.073
x5     -34.2938   14.781    -2.320    0.021    -63.300   -5.288
=====
Omnibus:             5.602   Durbin-Watson:      2.081
Prob(Omnibus):        0.061   Jarque-Bera (JB):   4.333
Skew:                 -0.036   Prob(JB):        0.115
Kurtosis:              2.686   Cond. No.      3.64e+03
=====

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 3.64e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
"""

```

### 6.3.8 Prediction

We can set some prediction values for this model:

TODO: fix this to reflect total reviews, not 0:1; Don't worry until we know where it goes.

### 6.3.8.1 R

```
# xp = seq(0, 1, by = .01)
xp = seq(-3, 3, by = .01)
Xp = splX(xp, knots)
```

### 6.3.8.2 Python

```
xp = np.arange(0, 1, 0.01)
Xp = splX(xp, knots)
```

While creating those predictions is nice, using them to visualize the model is far more helpful.

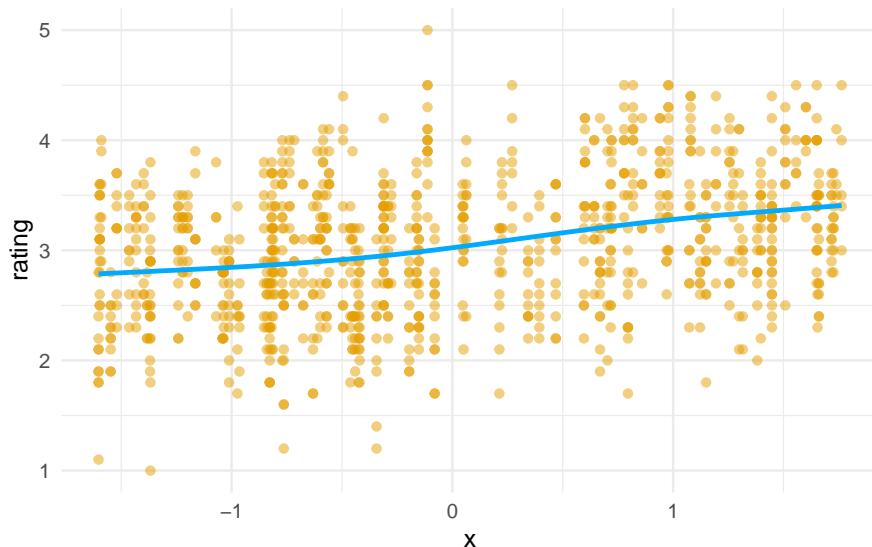


Figure 6.7: Visualizing cubic regression spline

In Figure 6.7, we can see that the relationship starts a bit flat, increases, and then flattens out again. This is a pretty good example of a nonlinear relationship.

### 6.3.9 Penalized Cubic Spline

Recall that this is an unpenalized cubic spline. If we want to have a finer degree of control over that wiggly line, we can include a **lambda penalty**.

We'll need to change up our spline function just a bit.

### 6.3.9.1 R

```
splS = function(knots) {
  q = length(knots) + 2
  S = matrix(0, q, q)
  S[3:q, 3:q] = outer(knots, knots, FUN = cubic_spline)
  S
}
```

### 6.3.9.2 Python

```
def splS(knots):
    q = len(knots) + 2
    S = np.zeros((q, q))
    S[2:, 2:] = cubic_spline(knots, knots[:,None])
    return S
```

We also need to be able to take the square root of our entire matrix. This is a bit more complicated than it sounds. We need to take the eigenvalue decomposition of the matrix, take the square root of the eigenvalues, and then recombine the matrix.

### 6.3.9.3 R

```
mat_sqrt = function(S) {
  d = eigen(S, symmetric = TRUE)
  rS = d$vectors %*% diag(d$values^.5) %*% t(d$vectors)
  rS
}
```

### 6.3.9.4 Python

```
def mat_sqrt(S):
    w, v = np.linalg.eig(S)
    rS = v @ np.diag(w**.5) @ v.T
    return rS
```

## 6.3.10 Penalized Model Fitting Function

With those functions in hand, we can create the function to fit the entire model.

### 6.3.10.1 R

```
prs_fit = function(y, x, knots, lambda) {
  q = length(knots) + 2    # dimension of basis
  n = length(x)            # number of observations
  Xa = rbind(splX(x, knots), mat_sqrt(splS(knots))*sqrt(lambda)) # augmented model matrix
```

```

y[(n + 1):(n+q)] = 0      # augment the data vector

lm(y ~ Xa - 1) # fit and return penalized regression spline
}

```

### 6.3.10.2 Python

```

def prs_fit(y, x, knots, lambda):
    q = len(knots) + 2
    n = len(x)
    Xa = np.vstack(
        (splX(x, knots), mat_sqrt(splS(knots))*np.sqrt(lambda)))
    )
    y_add = np.zeros(q)
    y = y.to_numpy()
    y = np.concatenate((y, y_add), axis = 0)
    return sm.OLS(y, Xa).fit()

```

Notice again that magic happens in the model matrix, but that we are still just using `lm` or `OLS` to fit the model.

## 6.3.11 Penalized Model Fitting

Let's stick with 4 knots and see what happens when we set our lambda to .1:

### 6.3.11.1 R

```

knots = 1:4/5

fit_penalized = prs_fit(
  y = rating,
  x = x,
  knots = knots,
  lambda = .1
)

```

```
Xp = splX(xp, knots)
```

### 6.3.11.2 Python

```

knots = np.arange(1, 5)/5

fit_penalized = prs_fit(
  y = y,
  x = x,
  knots = knots,
  lambda = .1
)

```

)

```
Xp = splX(xp, knots)
```

As shown in Figure 6.8, there is some wiggle to that line, but it is not as extreme as what we saw with our unpenalized cubic spline.

TODO: fix plot scale (doesn't match raw or scaled data)

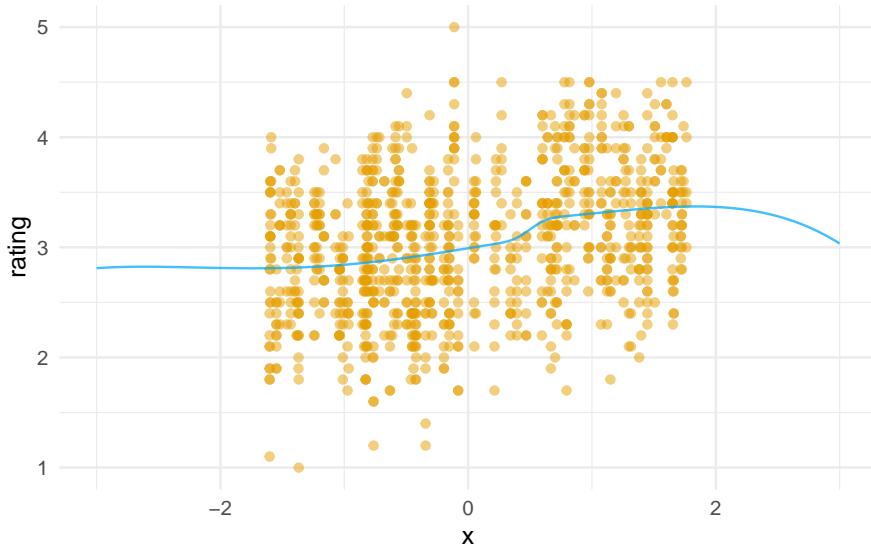


Figure 6.8: GAM model with lambda set to .1

We can test out what happens at different lambda values:

What can we take from Figure 6.9? As lambda values get closer to 1, we see lines that look very similar to a standard linear model. If you recall the our function to fit the model, we multiplied the square root of the matrix by the square root of the lambda value; since the square root of 1 is 1, we wouldn't see anything too interesting happen. As our lambda value gets lower, we see an increasing amount of wiggle happen.

Naturally, this is a great time to think about how these models would work on new data. As lambda gets smaller, we are fitting our in-sample data much better. How do you think this would fare with unseen data? If you'd say that we would do well with training and horrible on testing, we'd likely agree with you.

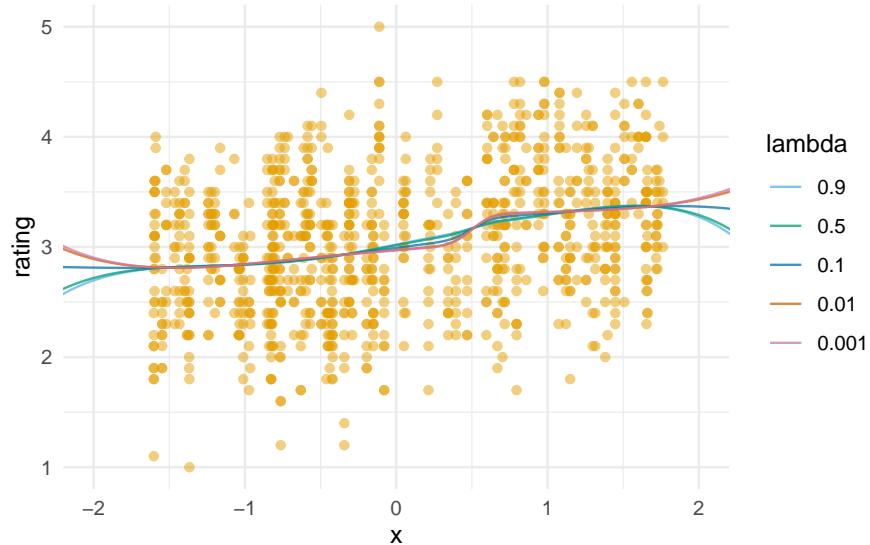


Figure 6.9: GAM model with different lambda values

## 6.4 Mixed Models

### 6.4.1 Why Should You Care?

Structures within your data are important and paying attention to how data might be grouped in some way will let you generate more insights about how observations within and between groups might behave. For example, people working in the same organization, or students in the same school, are likely to have a more similar experiences than those from different organizations or schools. This shared connection within groups may ultimately affect what we see in our chosen outcomes. Mixed models will let us handle grouped data such as this, all while getting the benefits of a standard linear model.

### 6.4.2 Knowing Your Data

As much fun as modeling is, knowing your data is far more important. You can throw any model you want at your data, from simple to fancy, but you can count on disappointment if you don't fundamentally know the structures that live within your data. Let's take a quick look at the following visualizations:

In Figure 6.10, we see a pretty solid positive relationship between the total number of reviews and ratings. We could probably just stop there, but we

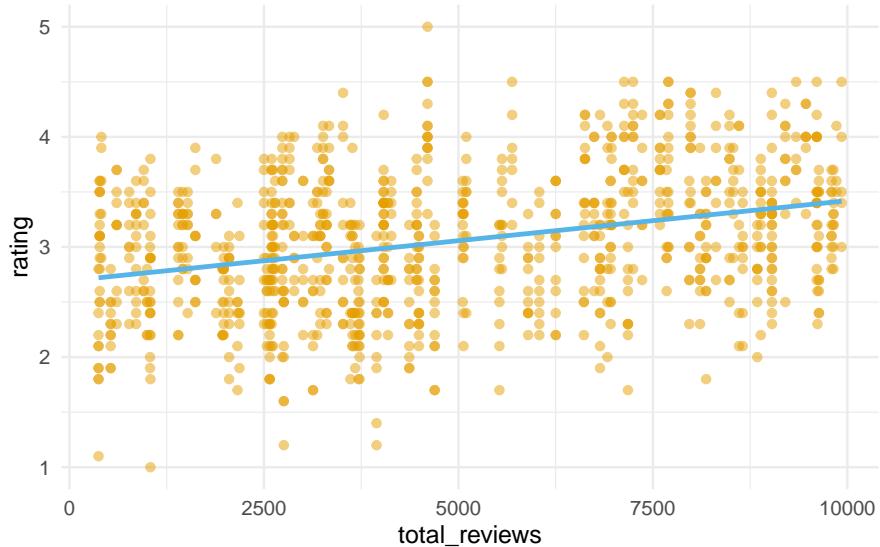


Figure 6.10: Linear relationship between year of movie release and rating.

might be ignoring something substantial within our data: genre. We might want to ask a question, “Does this relationship work the same way across the different genre?”

A very quick examination of Figure 6.11 might suggest that the relationship between user age and rating varies significantly over the different genres. Most genres show a positive relationship, while one (`other`) shows a negative relationship, and all with various levels of strength.

Just for fun, try to group by genre and summarize it by the mean rating. You’ll find that the averages are very different across the genre! Then, subtract the overall mean rating from those values, so that they represent deviations from the mean. Keep them handy!

Clearly, genre is offering some type of additional information to the model, but how can we incorporate that information into our model? An interaction might come to mind at first, and that’s the right way to think about it. A **mixed model** can be used to incorporate that information, which is largely a group interaction, into our model, without much hassle and with a huge boost in explanability.

The term **mixed model** is as vanilla as we can possibly make it, but you might have heard of different flavors of them before. You might have heard of *Hierarchical Linear Models*, or even *multilevel models*, or heard *mixed-effects models* tossed around before. Maybe you’ve even been exposed to ideas like

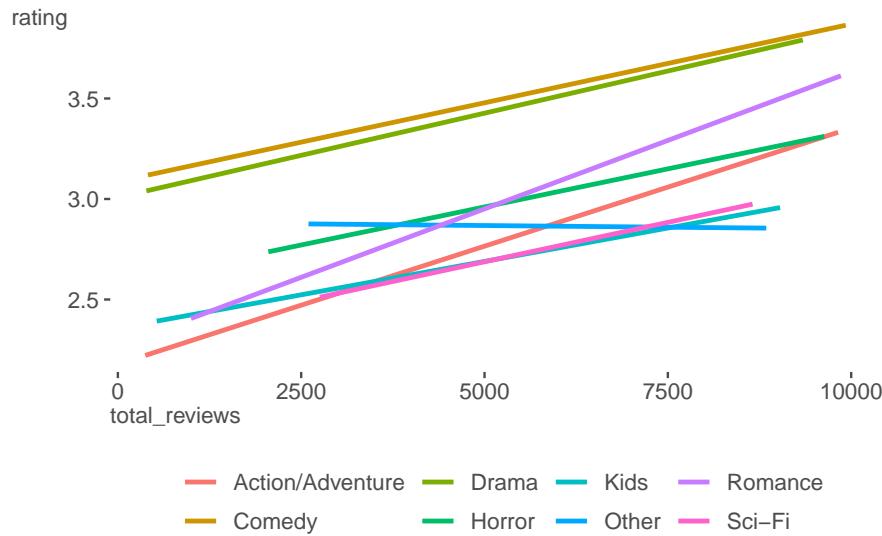


Figure 6.11: Linear relationship between year of movie release and rating, with genre.

*random effects* or *random slopes*. No matter what word you say, they are all instances of what we'll call a **mixed model**.

What makes a model a mixed model? The mixed model is characterized by the idea that a model can have *fixed* and *random* effects. Fortunately, you've already encountered *fixed* effects – those are the features that we have been using in all of our models so far! In the mixed model context, the *random effect*, typically comes from some type of grouping variable that contributes to unique variance in the outcome and we are going to let them vary from the rest of the model. Grouping variables can be actual groups, with the classic example being students in a classroom. Those groups can also be nested within larger groups – students nested within classrooms, nested within schools, nested within districts. These groups can also capture the notion of a repeated measure for an individual, like repeated test scores.

While we aren't rejecting the idea of a mean with these mixed-models, we are implying that each group (whether it is a group, nested group, or repeated observations from a person) has its own unique mean that can be useful for modeling the target. Formally, we might specify something like this:

$$\text{rating} = b_{\text{genre}} + b_{\text{year}} * \text{year}$$

We are explicitly saying that genre has its own unique effect for this model in

the form of specific intercepts for each genre. We also posit that those come from a *random* distribution. We can specify that as:

$$b_{0\_genre} \sim N(b_{intercept}, \sigma_{genre})$$

This means that the random intercepts will be normally distributed and the overall intercept is just the mean of those random intercepts, and with its own variance, an extra parameter we'll eventually have to estimate as part of the model. Another very common depiction is:

$$re\_group \sim N(0, \sigma_{genre})$$

$$b_{0\_genre} = b_{intercept} + re\_group$$

TODO: Need another example?

Before we go to modeling our reviews, let's consider an example training program to increase vertical jump, with average vertical increases of 2 inches. That really doesn't sound all that impressive; however, that increase came across 5 distinct groups: NBA players, NFL players, NHL players, MLB players, and data analysts. We can be pretty certain that each group has a very different vertical jump distribution coming into this training program. Given the amount of jumping that NBA players do, this program is unlikely to produce dramatic increases in vertical jump for them. We would probably expect modest gains in the NFL and even greater gains within NHL and MLB players. Where we are going to see the best gains, though, is from data analysts – they might want to jump to conclusions, but don't need to jump over their computers very often. Now that we know the additional information, can we just look at the average increase and be satisfied? Probably not. Instead, we need to look at the group that individuals might be in and judge accordingly. A mixed-model is going to let us to model all of this without too much work.

You might be asking? Why don't I just put genre into the model like other categorical features? In the case of genre, that's okay, but consider thousands of county voting percentages for an election, would you just put a county indicator into the model as is? Mixed models provide several advantages:

- Any coefficient can be allowed to vary by groups, including other random effects. It actually is just an interaction in the end.
- The group-specific effects are penalized, which shrinks them toward the overall mean. This helps to avoid overfitting, and that penalty is related to the variance estimate of the random effect. In other words, you're just running a penalized linear model where the penalty is applied to the group-specific effects.

- Standard modeling approaches only estimate the variance, and get the estimated group effects via a predictive method. This allows only one additional parameter to estimate rather than a weight or coefficient for every group.
- The group effects are like a very simplified **embedding**, where we have taken a categorical feature and turned it into a numeric one. This will help you understand techniques that are used in other places like deep learning.
- When you start to think about random effects and/or distributions for effects, you're already thinking like a Bayesian, who is always thinking about the distributions for various effects. Mixed models are a perfect segue from standard linear model estimation to Bayesian estimation.
- The random effect is akin to a latent variable of ‘unspecified group causes’. This is a very powerful idea that can be used in many different ways, but importantly, you might want to start thinking about how you can figure out what those ‘unspecified’ causes may be!
- Group effects will almost always improve your model’s performance relative to not having them, especially if you weren’t including those groups in your model because of how many there were.

In short, mixed models are a fun way to incorporate additional interpretive color to your model, while also getting several additional benefits!

### 6.4.3 Standard Functions

Now let’s fit some mixed models with `lme4` or `statsmodels`. We will also create `null models` (i.e., models with an intercept only), since we will need some information from them later.

#### 6.4.3.1 R

```
library(lme4)

fit_mer = lmer(rating ~ total_reviews_sc + (1 | genre),
               reviews,
               REML = FALSE)

summary(fit_mer)

Linear mixed model fit by maximum likelihood  ['lmerMod']
Formula: rating ~ total_reviews_sc + (1 | genre)
Data: reviews

      AIC      BIC    logLik deviance df.resid
1506.6   1526.2   -749.3    1498.6      996

Scaled residuals:
    Min     1Q Median     3Q    Max 
-1.50 -0.75 -0.25  0.25  1.50
```

```
-3.0992 -0.6840  0.0465  0.7187  3.1890

Random effects:
Groups   Name        Variance Std.Dev.
genre    (Intercept) 0.08309  0.2882
Residual           0.25482  0.5048
Number of obs: 1000, groups: genre, 8

Fixed effects:
            Estimate Std. Error t value
(Intercept)  2.9782     0.1038  28.69
total_reviews_sc 0.2479     0.0164  15.12

Correlation of Fixed Effects:
          (Intr)
ttl_rvws_sc -0.005

null_model = lmer(rating ~ 1 + (1 | genre),
                  reviews,
                  REML = FALSE)

summary(null_model, correlation = FALSE)

Linear mixed model fit by maximum likelihood  ['lmerMod']
Formula: rating ~ 1 + (1 | genre)
Data: reviews

      AIC      BIC      logLik deviance df.resid
1710.2   1724.9   -852.1    1704.2      997

Scaled residuals:
      Min       1Q       Median       3Q       Max
-3.08192 -0.76091 -0.04675  0.66740  2.98841

Random effects:
Groups   Name        Variance Std.Dev.
genre    (Intercept) 0.07557  0.2749
Residual           0.31371  0.5601
Number of obs: 1000, groups: genre, 8

Fixed effects:
            Estimate Std. Error t value
(Intercept)  2.98593   0.09962  29.98
```

### 6.4.3.2 Python

```
import statsmodels.api as sm

fit_mer = sm.MixedLM.from_formula("rating ~ total_reviews_sc", reviews,
                                    groups=reviews["genre"])

fit_mer = fit_mer.fit()

fit_mer.summary()

<class 'statsmodels.iolib.summary2.Summary'>
"""

      Mixed Linear Model Regression Results
=====
Model:           MixedLM  Dependent Variable: rating
No. Observations: 1000      Method:            REML
No. Groups:       8        Scale:             0.2551
Min. group size: 49      Log-Likelihood:     -753.8114
Max. group size: 300      Converged:          Yes
Mean group size: 125.0

-----
      Coef. Std.Err.    z   P>|z| [ 0.025  0.975]
-----
Intercept      2.978    0.111 26.863 0.000  2.761  3.195
total_reviews_sc 0.248    0.016 15.114 0.000  0.216  0.280
Group Var       0.095    0.104
-----
"""

null_model = sm.MixedLM.from_formula("rating ~ 1", reviews,
                                      groups=reviews["genre"])
```

What can we take from these summaries, and what are we getting beyond the linear model?

For starters, we should notice a change in our standard errors – by integrating information about the groups, we are getting a better sense of how much uncertainty our model contains at the global average level.

We also see some additional information for our random effects. The standard deviation is telling us how much the rating moves around based upon genre after getting the information from our fixed effects (i.e., the rating can move around nearly .3 points from genre alone).

We can use information for the random effects components to get some addi-

tional information. For instance, we can calculate the proportion of variance in scores that is accounted for by the genre alone:

$$\frac{\text{intercept variance}}{\text{intercept variance} + \text{residual variance}}$$

With our values, we have:

$$.08309 / (.08309 + .25482) = 0.2458939$$

This is what is known as the intraclass correlation (ICC). It ranges from 0 (no variance between clusters) to 1 (variance between clusters). With an ICC of .25, we can say that 25% of the variance in scores is accounted for by the genre alone.

**TODO: WEED PULLING**

We can also use various bits of information in our output to create different  $R^2$  values.

For the first level of the model, we can calculate the  $R^2$  as:

$$R_1^2 = \frac{\sigma_{M1}^2 + \tau_{M1}^2}{\sigma_{M0}^2 + \tau_{M0}^2}$$

We can pull that information from our two model outputs:

```
m1Sigma = .08309 # full model random effect intercept

m1Tau = .25482 # full model random effect residual

m0Sigma = .07557 # null model random effect intercept

m0Tau = .31371 # null model random effect residual

1 - ((m1Sigma + m1Tau) / (m0Sigma + m0Tau))

[1] 0.1319616
```

The fixed effect of number of reviews accounts for nearly 13% of the variation within the reviews.

The second level can be calculated as:

$$R_2^2 = 1 - \frac{\sigma_{M1}^2/B + \tau_{M1}^2}{\sigma_{M0}^2/B + \tau_{M0}^2}$$

We see that we added a  $B$  into the mix here and it is just the average size of the level 2 units (1000 observations / 8 genres).

```
level2Mean = 1000 / 8

r2Numerator = m1Sigma / level2Mean + m1Tau

r2Denominator = m0Sigma / level2Mean + m0Tau

1 - (r2Numerator / r2Denominator)

[1] 0.1871687
```

Which gives us .18 or 18% of the variation in scores is accounted for by the genre alone.

You can also get these values in R like this:

```
performance::r2(fit_mer)

# R2 for Mixed Models

Conditional R2: 0.362
Marginal R2: 0.154

MuMin::r.squaredGLMM(fit_mer)

R2m      R2c
[1,] 0.1539124 0.3619545
```

Here we have two values: the marginal R2 ( $R2m$ ) and the conditional R2 ( $R2c$ ). You can think of the marginal values as the standard type of R2 – it is the variability explained by the fixed effects part of the model (it is what we have already done above). The conditional R2 is using both fixed and random effects, so you can think of it as the total variability explained. Clearly, we can subtract the two to get a better understanding of the effect of the random effects. With  $.362 - .154 = .208$ , we can say that the random effects account for 20.8% of the variation in ratings.

Notice that those values are a bit different than what we produced by hand. Packages are now using a revised method proposed by Nakagawa, Johnson, and Schielzeth (2017) that is a bit more accurate than the previous method.

We can also get a good sense of the random effect estimates:

For Figure 6.12, the easiest way to think about it is that the values are effects for each individual random effect (i.e., each genre's random intercept). Since we are just dealing with intercepts right now, they are the deviations from the fixed intercept. The intercept for the model is  $\sim 2.6$  and the random effect for comedy is  $.48$ . If we wanted to predict scores for a comedy movie, we would take  $2.6 + .48$  for the intercept portion of the model (the same would go for any

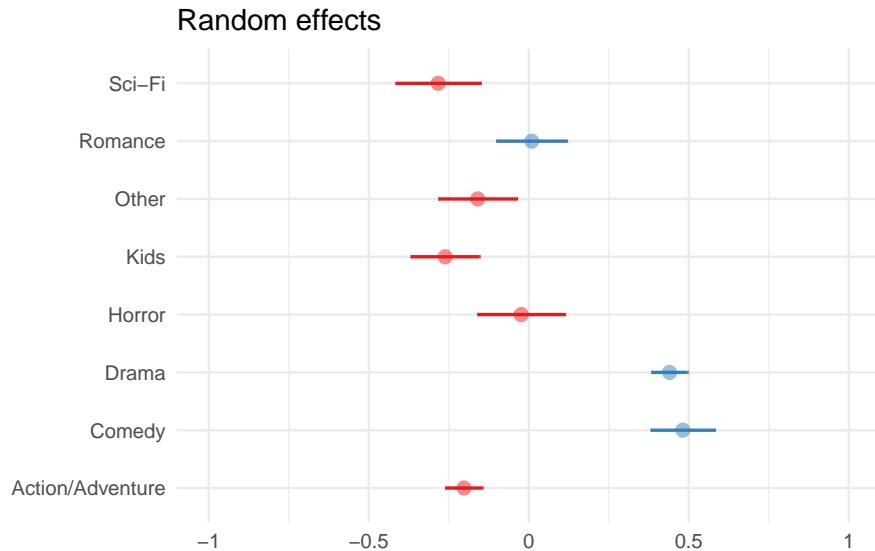


Figure 6.12: Random effects estimates

random slopes in the model). In the end, it is showing how much the intercept shifts from genre to genre, and some genres have a positive effect beyond the average and others have a negative effect (`sci-fi`, for instance, is well below the global average).

Remember your group-by and summarize task earlier? Each point is the difference between a genre's average and the overall average – values in blue are higher than the average and values in red are lower than the average. With that, the average rating for `comedy` is much better than the global average rating, while `sci-fi` is much worse than the global average rating.

#### 6.4.4 Model Matrix

Let's start our homebrewing adventures by creating a model matrix. We are going to use the `model.matrix()` function to create our model matrix. We are going to use the `user_age` variable as our fixed effect and `genre` as our random effect. We are going to use the `factor()` function to make sure that `genre` is treated as a categorical variable. We are also going to use the `-1` to remove the intercept from the model matrix.

##### 6.4.4.1 R

```
X = model.matrix(~total_reviews_sc, reviews)
Z = model.matrix(~factor(reviews$genre) - 1)
```

```
colnames(Z) = paste0("released_", sort(unique(reviews$genre)))

y = reviews$rating
```

#### 6.4.4.2 Python

```
X = reviews[['total_reviews_sc']]
X = sm.add_constant(X)
X = X.to_numpy()

Z = pd.get_dummies(reviews['genre'], drop_first=True)
Z = Z.to_numpy()

y = reviews['rating']
y = y.to_numpy()
```

#### 6.4.5 Likelihood Function

##### 6.4.5.1 R

```
mixed_log_likelihood = function(y, X, Z, theta) {
  tau = exp(theta[1])
  sigma = exp(theta[2])
  n = length(y)

  # evaluate covariance matrix for y
  e = tcrossprod(Z)*tau^2 + diag(n)*sigma^2
  b = coef(lm.fit(X, y))
  mu = X %*% b

  ll = mvtnorm::dmvnorm(y, mu, e, log = TRUE)
  -ll
}
```

##### 6.4.5.2 Python

```
from scipy import stats

def mixed_log_likelihood(theta, y, X, Z):
    tau = np.exp(theta[0])
    sigma = np.exp(theta[1])
    n = len(y)

    e = (Z.dot(Z.T) * tau**2) + (np.eye(n) * sigma**2)
    b = np.linalg.lstsq(X, y, rcond=None)[0]
```

```

mu = X.dot(b)

ll = stats.multivariate_normal.logpdf(y, mu, e)
return -ll

```

### 6.4.6 Model Fitting

#### 6.4.6.1 R

```

param_init = c(0, 0)

names(param_init) = c('tau', 'sigma')

fit = optim(
  fn  = mixed_log_likelihood,
  X   = X,
  y   = y,
  Z   = Z,
  par = param_init,
  # control = list(reltol = 1e-10)
)

exp(fit$par) # compare to sd of random effects

tau      sigma
0.2938593 0.5064417

```

#### 6.4.6.2 Python

```

from scipy import optimize as opt
def mixed_log_likelihood(theta, y, X, Z):
    tau = np.exp(theta[0])
    sigma = np.exp(theta[1])
    n = len(y)

    e = (Z.dot(Z.T) * tau**2) + (np.eye(n) * sigma**2)
    b = np.linalg.lstsq(X, y, rcond=None)[0]
    mu = X.dot(b)

    ll = stats.multivariate_normal.logpdf(y, mu, e)
    return -ll

theta = np.array([0, 0])

mixed_log_likelihood(theta, y, X, Z)

fit = opt.minimize(

```

```

    fun=mixed_log_likelihood,
    x0=theta,
    args=(y, X, Z),
    # tol=1e-5
)
np.exp(fit.x) # compare to sd of random effects

```

---

## 6.5 Performance Comparisons

Just for giggles, we should see how all of our models perform:

model	rmse
standard	0.59
median	0.59
gam	0.59
mixed	0.50

Figure 6.13: Comparing model performance with RMSE

Let's check out the results in Figure 6.13. Unsurprisingly, the standard linear model and the median regression were pretty close to each other. GAM offered a small bump in performance, but our best model came from the mixed model. This finding may or may not surprise you – as you spend more time with models, you often encounter situations where simple models outperform more complex models, or are on par with them. Here, we are seeing that the mixed model is offering us a better fit to the data than the other models. However, that doesn't mean that you can just go right to the mixed model. You need to know your data and know what you are trying to accomplish.

---

## 6.6 Wrapping Up

The standard linear model is useful across many different data situations. It does, unfortunately, have some issues when data becomes a little bit more “real”. When you have extreme scores or relationships that a standard model might miss, you don't need to abandon your linear model in favor of something more exotic. Instead, you might just need to think about how you are actually fitting the line through your data.

## 6.7 Next Steps

No matter how much we cover in this book, there is always more to learn. Here are some additional resources that you might find helpful related to this task. But if you've got a good grip on linear models and related topics, feel free to try out some machine learning Chapter 7!

If you want absolute depth on quantile regression, we will happily point you to the OG of quantile regression, Roger Koenker. His book, *Quantile Regression* is a must read for anyone wanting to dive deeper into quantile regression (2005b), or just play around with his R package `quantreg`. *Galton, Edgeworth, Frisch, and prospects for quantile regression in econometrics* is another resource from him.

If you want to dive more into the GAM world, we would recommend that you start with the **Moving Beyond Linearity** chapter in *An Introduction to Statistical Learning* (James et al. 2021). Not only do they have versions for both R and Python, but both have been made available online<sup>3</sup>. If you are wanting more after that, you can't beat Simon Wood's book, *Generalized Additive Models: An Introduction with R* (2017), or a more digestible covering of the same content by one of your own humble authors (Clark 2022).

There is no shortage of great references for mixed effects models. If you are looking for a great introduction to mixed models, we would recommend to start with yet another tutorial by one of your fearless authors! Michael Clark's *Mixed Models with R* (2023), is a great introduction to mixed models and is freely available<sup>4</sup>. If you want to dig just a little deeper, the `lme4` vignette for *Fitting Linear Mixed-Effects Models Using lme4*<sup>5</sup> is a great resource.

---

<sup>3</sup><https://www.statlearning.com/>

<sup>4</sup><https://m-clark.github.io/mixed-models-with-R/>

<sup>5</sup><https://cran.r-project.org/web/packages/lme4/vignettes/lmer.pdf>



---

---

## Part II

# Machine Learning



# 7

---

## Core Concepts

---

**Machine learning** is used everywhere, and allows us to do things that would have been impossible just a couple decades ago. It is used in everything from self-driving cars, to medical diagnosis, to predicting the next word in your text message. The ubiquity of it is such that it, and related adventures like artificial intelligence, are used as buzzwords, and it is not always clear what it meant by the one speaking them. In this chapter we hope you'll come away with a better understanding of what machine learning is, and how it can be used in your own work. Because however you define it, it sure can be fun!

Machine learning is a branch of data analysis with a primary focus on predictive performance. Honestly, that's pretty much it from a practical standpoint. It is not a subset of particular types of models, it does not preclude using statistical models, it doesn't mean that a program spontaneously learns without human involvement<sup>1</sup>, it doesn't necessarily have anything to do with 'machines' outside of laptop, and it doesn't even mean that the model is particularly complex. Machine learning, at its core, is a set of tools and a modeling approach that attempts to maximize and generalize performance, and compare models based on that performance<sup>2</sup>.

This is a *different focus* than statistical modeling approaches that put much more emphasis on interpreting coefficients and uncertainty. But it is *not an exclusive one*. Some implementations of machine learning include models that have their basis in traditional statistics, while others are often sufficiently complex that they are scarcely interpretable without a lot of effort, or are

---

<sup>1</sup>Although this is implied by the name, the description of ML as 'machines learning without human intervention' can be misleading to the newcomer. In fact, many of the most common models in machine learning are not capable of learning 'on their own' at any level, and require human intervention to provide processed data, specify the model, its parameters, set up the search through that parameter space, analyze the results, update the model, etc. We only very recently, post 2020, have developed models that appear to be able to generalize to new tasks as if they have learned them without human intervention, but that would ignore all the hands-on work that went into the development of those models, which never could have such capabilities otherwise.

<sup>2</sup>Generalization in statistical analysis is more about generalizing from our sample of data to the population from which it's drawn. In order to do that well or precisely, one needs to meet certain assumptions about the model. In machine learning, generalization is more about how well the model will perform on new data, and is often referred to as 'out-of-sample' performance.

used in contexts where interpretation simply isn't important. However, even after you conduct your modeling via machine learning, you may still fall back on statistical analysis for further exploration of the results. For example, you may want to know the uncertainty of your performance metric, or if the model is significantly better than another model. In any event, here we will also discuss some of the key ideas in machine learning, such as model assessment, loss functions, and cross-validation. Later we'll demonstrate common models used, but if you want to dive in, you can head there now!

**i ML by any other name...** AI, statistical learning, data mining, predictive analytics, data science, BI, there are a lot of names used alongside or even interchangeably with machine learning. It's mostly worth noting that using 'machine learning' without context makes it very difficult to know what tools have actually been employed, so you may have to do a bit of digging to find out the details.

## 7.1 Key ideas

- Machine learning is not a set of modeling techniques, but rather a modeling *focus* on predictive performance, and a set of tools and methods to achieve that.
- Models used in machine learning are typically more complex and difficult to interpret than those used in standard statistical models, but any model, including classical statistical ones, can be used with ML.
- There are many performance metrics used in machine learning, and care should be taken to choose the appropriate one for your situation.
- Objective functions likewise should be chosen for the situation, and are often different from the performance metric.
- Multiple performance metrics are able to be used for any given model assessment scenario.
- Regularization is a general approach to penalize complexity in a model, and is typically used to prevent overfitting in order to improve generalization.
- Cross-validation is a method that allows us to select parameters and hyperparameters for our models, and to compare models to one another by assessing a model's performance on data that was not used to fit the model.

### 7.1.1 Why this matters

Machine learning applications help define the modern world and how we interact with it. There are few aspects of modern society that have not been touched by it in some way. By understanding the basic ideas behind machine

learning, you will be able to understand the models and techniques that are used in these applications, and be able to apply them to your own work. You'll also be able to understand the limitations of these models.

### 7.1.2 Good to know

#### ADD LINKS

To dive into applying machine learning models, you really only need a decent grasp of linear models as applied to regression and classification problems. It would also be good to have an idea behind how they are estimated, as the same basic logic serves as a starting point here.

---

## 7.2 Objective Functions

We've implemented a variety of objective functions in other chapters, such as mean squared error for numeric targets and log loss for binary targets. As we have also noted elsewhere, the objective function is not necessarily the same as the performance metric we ultimately use to select a model. For example, we may use log loss as the objective function, but then use accuracy as the performance metric. In that setting, the log loss provides a 'smooth' objective function to search the parameter space over, while accuracy is a straightforward and more interpretable metric for stakeholders. In this case, the objective function is used to optimize the model, while the performance metric is used to evaluate the model. In some cases, the objective function and performance metric are the same (e.g. (R)MSE), and even if not, they might have selected the same 'best' model, but this is not always the case.

Table 7.1: Commonly used objective functions for standard regression and classification tasks

Objective Function	Description
Regression	
Mean Squared Error (MSE)	Calculates the average of the squared differences between predictions and actual values.
Mean Absolute Error (MAE)	Calculates the average of the absolute differences between predictions and actual values.
Huber Loss	Less sensitive to outliers than MSE.
Log Likelihood	Used for models where the response variable follows a known probability distribution.
Classification	
Binary Cross-Entropy / Log Likelihood (Loss)	Used for binary classification problems. Calculates the negative log likelihood of the predicted values.

---

Categorical Cross-Entropy

Used for multi-class classification problems. Calculates the

For specific types of tasks, such as predicting ranks, you might use something else, but the above will apply in some of the most common settings. Even when dealing with different types of outcomes, such as counts, proportions, etc., one can typically use a likelihood objective. Recall that ‘maximum likelihood’ functions can be turned into a minimization problem by taking the negative (log) likelihood.

---

### 7.3 Performance Metrics

There are many performance metrics used in machine learning, and care should be taken to choose the appropriate one for your situation. Typically we have a standard set we might use for the type of predictive problem. For example, for numeric targets, we typically are interested in (R)MSE and MAE, but given a particular scenario we might want to switch to something else. For example, if we are predicting a count type of target, we might use Poisson deviance, which is akin to a Poisson GLM in statistical modeling.

Most classification-based metrics are based on the **confusion matrix**, which is a table of the predicted classes versus the observed classes. Here is an example.

Table 7.2: Example Confusion Matrix

	Observed Negative	Observed Positive
Predicted Negative	62	10
Predicted Positive	10	18

The diagonal of the confusion matrix is the number of correct predictions, and the off-diagonal is the number of incorrect predictions. In this particular example we have an accuracy of 80 correct out of 100 total, or 80%. However, there are many metrics we can calculate from this simple confusion matrix<sup>3</sup>, and many of these can also be extended to the multiclass setting. For example, we can calculate the accuracy in general as well as for each class.

The following table illustrates the most commonly used metrics. Just because these are popular or applicable for your situation, doesn’t mean they are the only ones you can or even should use. Nothing keeps you from using more than one metric for assessment, and in fact, it is often a good idea to do so.

TODO: table needs work for pdf

---

<sup>3</sup>[https://en.wikipedia.org/wiki/Confusion\\_matrix](https://en.wikipedia.org/wiki/Confusion_matrix)

Table 7.3: Commonly used performance metrics in machine learning.

Metric	Description	Other Names/Notes
Regression		MSE (before square root)
RMSE	Root mean squared error	
MAE	Mean absolute error	
MAPE	Mean absolute percentage error	
RMSLE	Root mean squared log error	
R-squared	Amount of variance shared by predictions and observed target	Coefficient of determination
Deviance/AIC	Generalization of sum of squared error for non-continuous/gaussian settings	Also "deviance explained" for similar R-sq inter
Classification		
Accuracy	Percent correct	Error rate is 1 - Accuracy
Precision	Percent of positive predictions that are correct	Positive Predictive Value
Recall	Percent of positive samples that are predicted correctly	Sensitivity, True Positive Rate
Specificity	Percent of negative samples that are predicted correctly	True Negative Rate
Negative Predictive Value	Percent of negative predictions that are correct	
F1	Harmonic mean of precision and recall	F-Beta <sup>1</sup>
AUC	Area under the ROC curve	Type I Error, alpha
False Positive Rate	Percent of negative samples that are predicted incorrectly	Type II Error, beta, Power is 1 - beta
False Negative Rate	Percent of positive samples that are predicted incorrectly	
Phi	Correlation between predicted and actual	Matthews Correlation
Log loss	Negative log likelihood of the predicted probabilities	

<sup>1</sup>Beta = 1 for F1

As an example, and as a reason to get our first taste of machine learning, let's get some additional metrics for our movie review model. Depending on the tool used, getting one type of metric should be as straightforward as most others if we're using common metrics. As we start our journey into machine learning, we'll show Python code first, as it's the dominant tool.

### 7.3.0.1 Python

In Python, we can use the `sklearn.metrics` module to get a variety of metrics for both regression and classification problems.

```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.metrics import roc_auc_score, roc_curve, auc, confusion_matrix

from sklearn.linear_model import LinearRegression, LogisticRegression

import pandas as pd

df_movie_reviews = pd.read_csv("data/movie_reviews_processed.csv")

X = df_movie_reviews[
    [
        'word_count',
        'age',
        'review_year',
        'release_year',
        'length_minutes',
        'children_in_home',
        'total_reviews',
    ]
]

y = df_movie_reviews['rating']
y_class = df_movie_reviews['rating_good']

model_lin_reg = LinearRegression()
model_lin_reg.fit(X, y)

# note that sklearn uses regularization by default for logistic regression
model_log_reg = LogisticRegression()
model_log_reg.fit(X, y_class)

y_pred_linreg = model_lin_reg.predict(X)
y_pred_logreg = model_log_reg.predict(X)
```

```
# regression metrics
rmse = mean_squared_error(y, y_pred_linreg, squared=False)
mae = mean_absolute_error(y, y_pred_linreg)
r2 = r2_score(y, y_pred_linreg)
```

```
# classification metrics
accuracy = accuracy_score(y_class, y_pred_logreg)
precision = precision_score(y_class, y_pred_logreg)
recall = recall_score(y_class, y_pred_logreg)
```

### 7.3.0.2 R

In R, we can use the `yardstick` package, which has a consistent interface for a variety of metrics.

```
library(yardstick)

# convert rating_good to factor for yardstick input
df_movie_reviews = read_csv("data/movie_reviews_processed.csv") |>
  mutate(rating_good = factor(rating_good, levels = c(0, 1), labels = c("bad", "good")))

model_lin_reg = lm(
  rating ~
    word_count
    + age
    + review_year
    + release_year
    + length_minutes
    + children_in_home
    + total_reviews,
  data = df_movie_reviews
)

model_log_reg = glm(
  rating_good ~
    word_count
    + age
    + review_year
    + release_year
    + length_minutes
    + children_in_home
    + total_reviews,
  data = df_movie_reviews,
  family = binomial(link = "logit")
)
```

```

y_pred_linreg = predict(model_lin_reg)
y_pred_logreg = predict(model_log_reg, type = "response")
y_pred_logreg = factor(ifelse(y_pred_logreg > .5, "good", "bad"))

# regression metrics
rmse = rmse_vec(df_movie_reviews$rating, y_pred_linreg)
mae = mae_vec(df_movie_reviews$rating, y_pred_linreg)
r2 = rsq_vec(df_movie_reviews$rating, y_pred_linreg)

# classification metrics
accuracy = accuracy_vec(df_movie_reviews$rating_good, y_pred_logreg)
precision = precision_vec(df_movie_reviews$rating_good, y_pred_logreg)
recall = recall_vec(df_movie_reviews$rating_good, y_pred_logreg)

```

We put them all together in the following table. Now we know how to get them, and it was easy! But as we'll see later, there is a lot more to think about before we use these for model assesment.

Table 7.4: Demo Metrics

Metric	Value
<hr/>	
Linear Regression	
RMSE	0.52
MAE	0.41
R-squared	0.32
<hr/>	
Logistic Regression	
Accuracy	0.71
Precision	0.69
Recall	0.60

---

## 7.4 Generalization

One of the key differences separating ML from traditional statistical modeling approaches is the assessment of performance on unseen or future data, a concept commonly referred to as **generalization**. The basic idea is that we want to build a model that will perform well on new data, and not just the data we used to fit the model, because ultimately data is ever evolving, and

we don't want to be beholden to a particular set of data we just happened to have at a particular time.

But how do we do this? For starters, we can simply split our data into two sets, a **training set** and a **test set**, often called a **holdout set**. The test set is typically a smaller subset, say 25%, but this amount is arbitrary, and will reflect the data situation. We **fit** the model on the training set, and then use the model to make predictions on, or **score**, the test set. This general approach is also known as the **holdout method**. Consider a simple linear regression. We can fit the linear regression model on the training set, which provides us coefficients, etc. We can then use that model result to predict on the test set, and then compare the predictions to the actual values in the test set. Here we demonstrate this with our simple linear model from before.

#### 7.4.0.1 Python

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import pandas as pd

X = df_movie_reviews[
    [
        'word_count',
        'age',
        'review_year',
        'release_year',
        'length_minutes',
        'children_in_home',
        'total_reviews',
    ]
]

y = df_movie_reviews['rating']

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=123
)

model = LinearRegression()
model.fit(X_train, y_train)

# get predictions
y_pred = model.predict(X_test)

# get RMSE on test
```

```
mean_squared_error(y_test, y_pred, squared=False)
```

```
RMSE on test: 0.53
```

#### 7.4.0.2 R

```
# create a train and test set
library(rsample)

set.seed(123)

split = initial_split(df_movie_reviews, prop = .75)

df_train = training(split)
df_test = testing(split)

model_reviews_extra = lm(
  rating ~
  word_count
  + age
  + review_year
  + release_year
  + length_minutes
  + children_in_home
  + total_reviews,
  data = df_train
)

# get predictions
test_preds = predict(model_reviews_extra, newdata = df_test)

# get RMSE on test
yardstick::rmse_vec(df_test$rating, test_preds)

RMSE on test: 0.54
```

So there you have it, we have a model that we can use to predict on new data without too much trouble. As we'll soon see though, there are limitations to doing things this simply. But conceptually this is an important idea, and one we will continue to return to in our discussion of machine learning.

#### 7.4.1 Using Metrics for Model Evaluation and Selection

As we've seen, there are many performance metrics to choose from to assess model performance, and the choice of metric depends on the type of problem. For example, for a problem for numeric targets, we might use RMSE, while for a classification problem, we might use accuracy. As discussed, it turns out

that assessing the metric on the data we used to fit the model does not give us the best assessment of that metric. This is because the model will do better on the data it was trained on than on new data it wasn't trained on, and we can generally always improve that metric in training by making the model more complex. However, in many modeling situations, this complexity comes at the expense of generalization, and the model will not perform as well on new data, something we'll discuss in more detail shortly. So what we really want to ultimately say about our model will regard performance on the test set with our chosen metric, and not the data we used to fit the model. At that point, we can also compare multiple models to one another given their performance on the test set, and select the one that performs best.

You should take a moment to compare our result with the holdout method to the initial model we used to get some metrics, where the model was fit on the entire dataset. Metrics are almost always better on the training set, and that's because the model was fit on the entire dataset, and so it was able to capture more of the variability in the data. But that's not what we're interested in. We want to know how well the model will do on new data, and so we use the test set to get a sense of that.

### 7.4.2 Understanding Test Error and Generalization

This part gets into the weeds a bit. If you are not so inclined, skip to the summary of this section.

In the following discussion, you can think of a standard linear model scenario just as we have been using, e.g. with squared-error loss function, and a data set where we split some of the observations in a random fashion into a training set, for initial model fitting, and a test set, which will be kept separate and independent, and used to measure generalization performance. We note **training error** as the average loss over all the training sets we could create in this process, and **test error** as the average prediction error obtained when a model fitted on the training data is used to make predictions on the test data. So, in addition to the previously noted goal of finding the 'best' model, **model selection**, we are interested further in estimating the prediction error with new data, **model performance**.

#### 7.4.2.1 Generalization in the Classical Regime

So consider a modeling situation where we have the usual situation of splitting data into training and test sets. We run the model on the training set, but we are more interested in generalization error, or how well it predicts on the test set. We can think of the test error as the average error over many such splits of the data into training and test sets. Given this scenario, let's look at the following visualization inspired by Hastie, Tibshirani, and Friedman (2017).

Prediction error on the test set, shown in red, is a function of several compo-

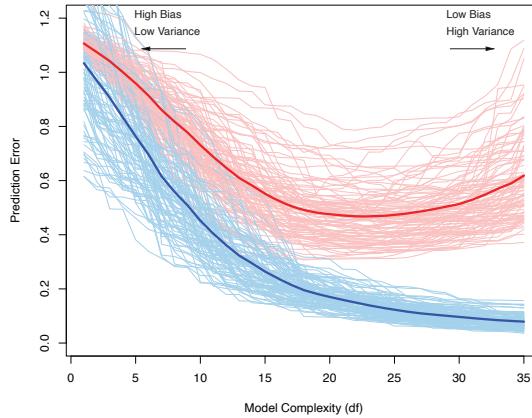


Figure 7.1: Bias Variance Tradeoff

nents, and the terms bias and variance generally refer to two of those components. One thing to note is that even if we had the ‘true’ model given the features specified correctly, there would still be prediction error due to the random data generating process.

The main idea here is that as the model complexity increases, we potentially capture more of the data variability. The so-called **bias**, which is the difference in our average prediction and the true model prediction, decreases, but this only continues for training error, shown in blue, where eventually our model can possibly fit the training data perfectly! For test error though, as the model complexity increases, the bias decreases, but the **variance**, which is the variability in prediction with changes in data, eventually increases. This is because we get too close to the training data and do poorly when we try to generalize beyond it. This is traditionally known as the **bias-variance tradeoff** - we can reduce one source of error in the test set at the expense of the other, but not both at the same time indefinitely. In other words, we can reduce bias by increasing model complexity, but this will eventually increase variance in our test predictions. We can reduce variance by reducing model complexity, but this will increase bias. The goal is to find the sweet spot where we have a model that is complex enough to capture the underlying process, but not so complex that it overfits to the training data. Recall that we’re not as interested in training error except to get a sense of how well the model fits the data- ideally it at least does well on training!

#### 7.4.2.2 Generalization in Deep Learning

It turns out that with lots of data and very complex models, or maybe just in most settings, our classical understanding doesn’t hold up like we’d think. In fact, we can get a model that fits the training data perfectly, and yet ultimately

still generalizes well to new data! This phenomenon is encapsulated in the notion of **double descent**. The idea is that, with overly complex models such as those employed with deep learning, we get to the point of interpolating the data exactly (Figure 7.3). But as we continue to increase the complexity of the model, we actually start to generalize better again, and visually this displays as a double descent in terms of test error. We see an initial decrease in test error as the model gets better in general. After a while, it begins to rise as seen in the classical regime (Figure 7.1), to where we hit a peak at the point where we have as many parameters as data points. Beyond that however, as we go even more complex with our model, we can possibly see a decrease in test error again. Crazy!

We demonstrate this on the classic `mtcars` dataset<sup>4</sup>, which has only 32 observations! We repeatedly train a model to predict miles per gallon on only 10 of those observations, and assess test error on the rest. The model we use is a form of ridge regression, but implemented such that we can use splines for the car's weight, horsepower, and displacement<sup>5</sup>. We fit increasingly complex models, and plot the test error and training error as a function of model complexity. We see that the test error dips as we get a better model, but eventually rises as expected. It eventually hits a peak, but then starts to decrease again! This is the double descent phenomenon with one of the simplest datasets around. Cool!

#### 7.4.2.3 Generalization Summary

The take home point is this: our primary concern is generalization error. We can reduce this error by increasing model complexity, but this may eventually cause test error to increase. However, with enough data and model complexity, we can get to the point where we can fit the training data perfectly, and yet still generalize well to new data. Unless you are doing deep learning, you can maybe assume the classical regime holds, but when doing deep learning, you can worry less about the model's complexity. In any event, we still want to employ tools to help reduce generalization error, and we prefer smaller and simpler models that can do as well as more complex ones, even if those models are still billions of parameters!

---

## 7.5 Regularization

As we've seen, a key aspect of the machine learning approach is to generalize to new data. One way to improve generalization is through the use of `reg-`

<sup>4</sup>If not familiar, the `mtcars` object is a data frame that comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models).

<sup>5</sup>It's actually called <sup>6</sup>

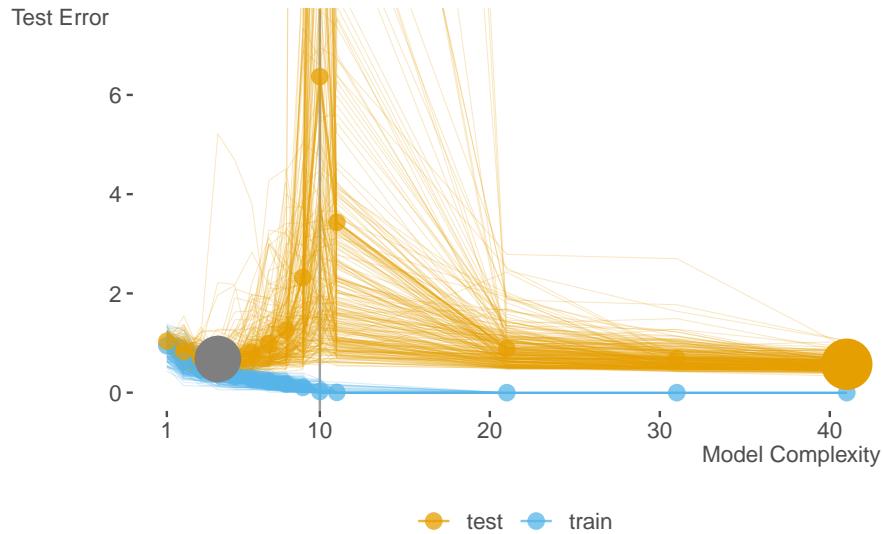


Figure 7.2: Double Descent on the classic mtcars dataset

**ularization**, which is a general approach to penalize complexity in a model, and is typically used to prevent **overfitting**. Overfitting occurs when a model fits the training data very well, but does not generalize well to new data, and this is often due to the model being too complex for the data setting, and is fitting to noise in the training data that isn't present in other data. Note that the converse can also happen, and is often the case with simpler models, where the model does not fit the training data well, and so does not generalize well to new data either, and this is known as **underfitting**<sup>7</sup>.

We demonstrate this in the following visualization. The first plot shows results from a model that is notably complex, and in doing so presents a very wiggly result. This is an example of overfitting, and is often seen in models that are too complex for the underlying data. The second plot shows a straight line fit as we'd get from linear regression, which is an example of underfitting. The third plot shows a model that is a better fit to the data, and is an example of a model that is complex enough to capture the nonlinear aspect of the data, but not so complex that it is trying to capitalize on noise in the data.

<sup>7</sup>Underfitting is a notable problem in many academic disciplines, where the models are often too simple to capture the complexity of the underlying process. Typically the models are often linear, and the underlying process may be anything but. These disciplines were slow to adopt machine learning techniques, as they are often more difficult to interpret, and so seen as not as useful for understanding the underlying process. However, one could make the obvious argument that 'understanding' an unrealistic result is not very useful either, and that the goal should be to understand the underlying process however we can, and not just the model.

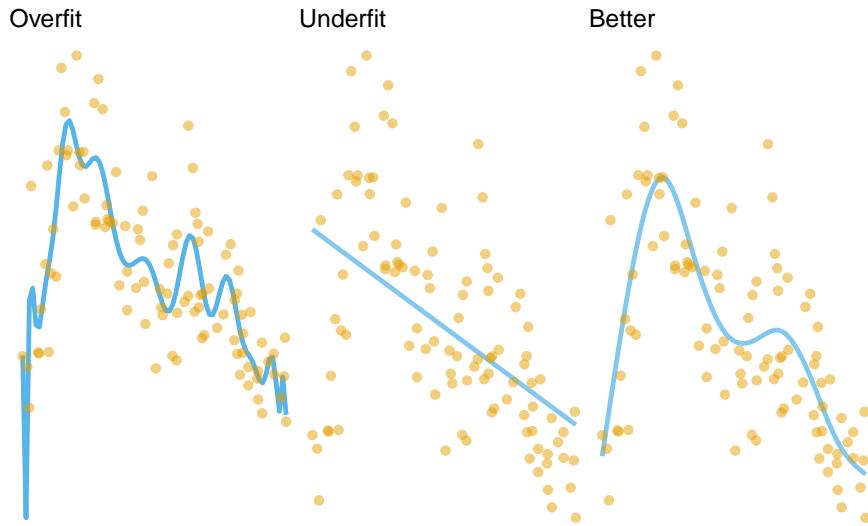


Figure 7.3: Overfitting and Underfitting

When we examine generalization performance<sup>8</sup>, we see that the overfit model does best on training data, but relatively very poorly on test- nearly a 20% increase in the RMSE value. The underfit model doesn't change as much in performance because it was poor to begin with on training. Our 'better' model wasn't best on training, but was best on the test set.

Table 7.5: RMSE for each model on new data

Model	RMSE	% change
<hr/>		
Train		
Better	2.18	
Over	1.97	
Under	3.05	
<hr/>		
Test		
Better	2.19	0.6
Over	2.34	19.1
Under	3.24	6.1

<sup>8</sup>The data is based on a simulation (using `mgcv::gamSim`), so the test data is just more simulated data points.

We have already seen one example of regularization in the ridge regression model (ADD CHAPTER LINK), where we add a penalty term to the objective function. This penalty term is a function of the coefficients, and is based on the sum of the squared values of the coefficients. It is also known as an **L2 penalty**, and is a very common type of regularization. Another common approach for linear models is the **L1 penalty**, which is the sum of the absolute values of the coefficients. This is used in the **lasso** model. There are other types of regularization as well, such as the elastic net, which is a combination of the L1 and L2 penalties. The relative size of the two penalties is controlled by a mixing parameter, and the optimal value of that parameter is determined by cross-validation.

It turns out that regularization is used in many modeling scenarios. Here is a quick rundown of some examples.

- GAMs also use penalized regression for estimation, where the coefficients used in the basis functions are penalized (typically with L2). This keeps the ‘wiggly’ part of the GAM from getting too wiggly, as in the overfit model above (Figure 7.3), tending toward a linear effect.
- Similarly, the variance estimate of a random effect in mixed models, e.g. for the intercept or slope, is inversely related to an L2 penalty on the fixed effects estimates for that group effect. The more penalization applied, the less random effect variance, and the more the random effect is shrunk toward the overall mean<sup>9</sup>.
- Still another form of regularization occurs in the form of priors in Bayesian models. For example, the variance on the prior for regression coefficients could be very large, which amounts to a result where there is little influence of the prior on the posterior, or it could be very small, which amounts to a result where the prior has a lot of influence on the posterior, shrinking it toward the prior mean, which is typically zero. In fact, ridge regression is a frequentist form of standard Bayesian linear regression with a normal distribution prior for the coefficients, and the L2 penalty is related to the variance of that prior.
- As a final example of regularization, **dropout** is a technique used in deep learning to prevent overfitting. It works by randomly dropping out some of the nodes in intervening/hidden layers in the network during training. This tends to force the network to learn more robust features, allowing for better generalization.

In short, regularization comes in many forms across the modeling landscape, and is a key aspect of machine learning and traditional statistical modeling alike. In general, we can add the same sort of penalty to any number of

---

<sup>9</sup>One more reason to prefer a random effects approach over so-called fixed effects models, as the latter are not penalized at all, and thus are more prone to overfitting.

models, such as logistic regression, neural networks, recommender systems etc. The primary goal again is to hopefully increase our ability to generalize the selected model to new data.

---

## 7.6 Cross-validation

So we've talked a lot about generalization to unseen data, so now let's think about some ways to go about a general process of selecting parameters for a model and assessing performance and generalization.

As noted previously, the simplest approach is to split the data into training and test sets, fit the model on the training set, and then assess performance on the test set. This is all well and good, but the test error has uncertainty, and would be slightly different with any training-test split we came up with. We'd also like to get a better assessment when searching the parameter space, because there are oftentimes parameters for which we have no way of guessing the value beforehand. In this case, we need to figure out the best parameters *before* assessing a final model's performance. One way to do this is to split the data into multiple test sets, which we now call **validation sets**, because we still want a test set to be held out that is in no way used during the training process. We fit the model on the training set, and then assess performance on the validation set(s). We then repeat this process for many different splits of the data into training and validation sets, and average the results. This is known as **K-fold cross-validation**.

Here is a visualization of 3-fold cross validation. We split the data such that 2/3 of it will be used for training, and 1/3 for validation. We then do this for a total of 3 times, such that the validation set is on a different part of the data each time, and all observations are used for both training and validation at some point. We then average the results of any metric across the validation sets. Note that in each case here, there is no overlap of data between the training and validation sets.



The idea is that we are trying to get a better estimate of the test error by averaging over many different test sets. The number of folds, or splits, is denoted by  $K$ . The value of  $K$  can be any number, but typically is 10 or less. The larger the value of  $K$ , the more accurate the estimate of the test error, but the more computationally expensive it is, and in application, you generally don't need much to get a good estimate of the mean error. However, with smaller datasets, one can even employ a **leave-one-out** approach, where  $K$  is equal to the number of observations in the data.

So cross-validation provides a better measure of the test error. If we are interested when we look at models with different parameters we're trying to figure out, we can pit their respective average errors against one another, and select the model with the lowest average error, a process known generally as **model selection**. This works for choosing a model within a potential set of hyperparameter settings, for example, with different penalty parameters for regularized regression, but can also aid in choosing a model from a set of different model types, for example, standard linear model approach vs. boosting.

Now how might we go about this for modeling purposes? Very easily with modern packages. In the following we demonstrate this with a logistic regression model.

### 7.6.0.1 Python

```
# import necessary libraries
```

```

from pandas import read_csv
from sklearn.linear_model import LogisticRegressionCV
from sklearn.metrics import accuracy_score

df_movies = read_csv("data/movie_reviews_processed.csv")

X = df_movies.filter(regex="_sc$")
y = df_movies["rating_good"]

# Cs is the (inverse) penalty parameter;
clf = LogisticRegressionCV(penalty='l2', Cs=[1], cv=5, max_iter=1000)
clf.fit(X, y)

LogisticRegressionCV(Cs=[1], cv=5, max_iter=1000)

# clf.scores_ # show the accuracy score for each fold

# print the average accuracy score
clf.scores_[1].mean()

0.671

```

### 7.6.0.2 R

For R, we prefer `mlr3` for our machine learning demonstrations, as we feel it is more like `sklearn` in spirit, as well as offering computational advantages for when you want to actually do ML with R. The `tidymodels` ecosystem is also a good option.

```

# Load necessary libraries
library(mlr3)
library(mlr3learners)

df_movies = read_csv(
  "data/movie_reviews_processed.csv",
  col_select = matches('_sc|rating_good')
)

df_movies = df_movies %>%
  mutate(rating_good = as.factor(rating_good))

# Define task
task_lr_ridge = TaskClassif$new("movie_reviews", df_movies, target = "rating_good")

# Define learner (alpha = 0 is ridge regression)
learner_lr_ridge = lrn("classif.cv_glmnet", alpha = 0, predict_type = "response")

```

```

learner_lr_ridge$param_set$values$alpha = 1 # set the penalty parameter to some value

# Define resampling strategy
result_lr_ridge = resample(
  task      = task_lr_ridge,
  learner   = learner_lr_ridge,
  resampling = rsmp("cv", folds = 5)
)

INFO [17:16:57.236] [mlr3] Applying learner 'classif.cv_glmnet' on task 'movie_reviews' (iter 1/5)
INFO [17:16:57.395] [mlr3] Applying learner 'classif.cv_glmnet' on task 'movie_reviews' (iter 2/5)
INFO [17:16:57.452] [mlr3] Applying learner 'classif.cv_glmnet' on task 'movie_reviews' (iter 3/5)
INFO [17:16:57.507] [mlr3] Applying learner 'classif.cv_glmnet' on task 'movie_reviews' (iter 4/5)
INFO [17:16:57.564] [mlr3] Applying learner 'classif.cv_glmnet' on task 'movie_reviews' (iter 5/5)

# result_lr_ridge$score(msr('classif.acc')) # show the accuracy score for each fold

# print the average accuracy score
result_lr_ridge$aggregate(msr('classif.acc'))

classif.acc
0.675

```

In each case above, we end up with five separate accuracy values, one for each fold. Our final assessment of the model's accuracy is the average of these five values. This is a better estimate of the model's accuracy than if we had just used a single test set, and in the end it is based on the entire data.

### 7.6.1 Methods of Cross-validation

There are different approaches we can take for cross-validation that we may need for different data scenarios. Here are some of the more common ones.

- **Shuffled:** Shuffling prior to splitting can help avoid data ordering having undue effects.
- **Grouped/stratified:** In cases where we want to account for the grouping of the data, e.g. for data with a hierarchical structure. We may want groups to appear in training *or* test, but not both, as with grouped k-fold. Or we may want to ensure group proportions across training and test sets, as with stratified k-fold.
- **Time-based:** e.g. for time series data, where we only want to assess error on future values
- **Combinations:** e.g. grouped and time-based

Here are images from the scikit-learn library documentation<sup>10</sup> depicting some different cross-validation approaches.

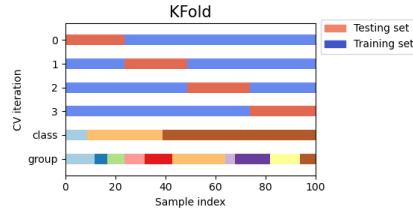


Figure 7.4: k-fold

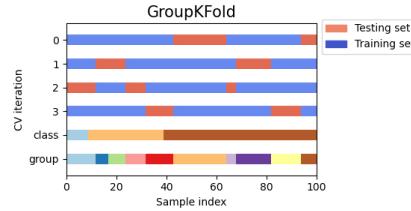


Figure 7.5: Grouped

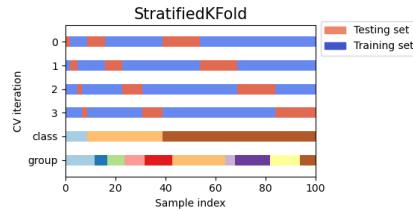


Figure 7.6: Stratified

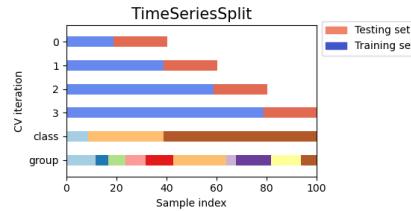


Figure 7.7: Time series

In general, the form we employ will be based on our data needs.

**💡** It's generally always useful to use a stratified approach to cross-validation, especially with classification problems, as it helps ensure a similar balance of the target classes across training and test sets. You can also employ this with numeric targets, enabling you to have a similar distribution of the target across training and test sets.

## 7.7 Tuning

One problem with the previous ridge logistic model we just used is that we set the penalty parameter to a fixed value. We can do better by searching over a range of values instead, and picking a ‘best’ one. This is generally known as **hyperparameter tuning**, or simply **tuning**. We can do this with cross-validation as well where we will use k-fold cross-validation to assess the error for each value of the penalty parameter values. We then select the value of the penalty parameter that gives the lowest average error. This is a form of **model selection**.

<sup>10</sup>[https://scikit-learn.org/stable/auto\\_examples/model\\_selection/plot\\_cv\\_indices.html](https://scikit-learn.org/stable/auto_examples/model_selection/plot_cv_indices.html)

Another potential point of concern is that we are using the same data to both select the model and assess its performance. This is a form of a more general phenomenon of **data leakage**, and may result in an overly optimistic assessment of performance. One solution is to do as we've discussed before, which is to split the data into three parts: training, validation, and test. We use the training set(s) to fit the model, the validation set(s) to select the model, and then finally use the test set to assess the model's performance. The validation approach is used to select the model, and the test set is used to assess the model's performance. The following visualizations from the scikit-learn documentation illustrates the process.

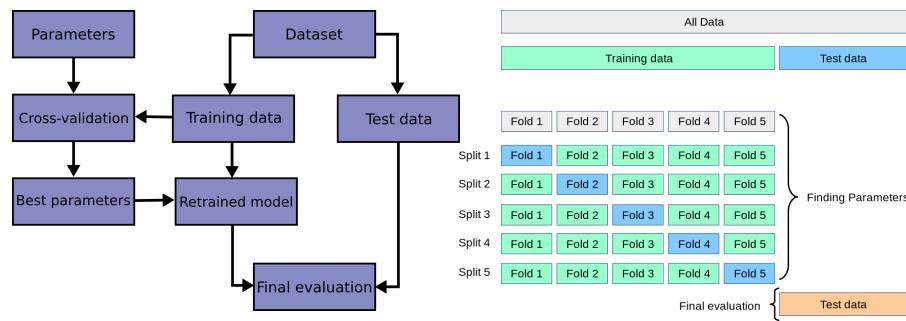


Figure 7.8: Train-Validation-Test Workflow

**i** As the performance on test is not without uncertainty, we can actually nest the entire process within a validation approach, where we have an inner loop of k-fold cross-validation and an outer loop to assess the model's performance on multiple hold out sets. This is known as **nested cross-validation**. This is a more computationally expensive approach, and generally would require more data, but it would result in a more robust assessment of performance

### 7.7.1 A Tuning Example

While this may start to sound complicated, it doesn't have to be, as tools are available to make our generalization journey a lot easier. In the following we demonstrate this with a ridge based logistic regression model. The approach we use is called a **grid search**, where we explicitly step through potential values of the penalty parameter. While we only look at one parameter here,

for a given modeling approach we could construct a ‘grid’ of sets of parameter values<sup>11</sup> to search over as well.

We use the `LogisticRegression` function in `sklearn` to perform k-fold cross-validation to select the best penalty parameter. We then apply the best model to the test set and calculate accuracy. We do the same thing in R with the `mlr3tuning` package. We use the `AutoTuner` function to perform k-fold cross-validation to select the best penalty parameter. In both settings we are interested in the average accuracy score across the folds, and ultimately the test set<sup>12</sup>.

### 7.7.1.1 Python

```
# import necessary libraries
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

X = df_movie_reviews.filter(regex="_sc$")
y = df_movie_reviews["rating_good"]

# split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.25,
    random_state=42
)

# define the parameter values for GridSearchCV
param_grid = {
    'C': [0.1, 1, 2, 5, 10, 20],
}

# perform k-fold cross-validation to select the best penalty parameter
# Note that LogisticRegression by default is ridge regression for scikit-learn
grid_search = GridSearchCV(
```

---

<sup>11</sup>We can use of `expand.grid` or `crossing` in R, or pandas’ version `expand_grid` to construct these values to iterate over. scikit-learn’s `GridSearchCV` function does this for us when we provide the dictionary of values for each parameter.

<sup>12</sup>If you’re comparing the Python vs. R approaches, scikit-learn by default uses ridge regression, while in R we set the value alpha to enforce it, since `glmnet` by default uses the elastic net, a mixture of lasso and ridge. Also, scikit-learn uses the inverse of the penalty parameter, while `mlr3` uses the penalty parameter directly, which is more straightforward. And obviously, no one will agree on what we should name the value (we have no idea where ‘C’ comes from, maybe ‘complexity’(?), though we have seen `alpha` used in various statistical publications).

```
LogisticRegression(), param_grid=param_grid, cv=5, scoring='accuracy'
)

grid_search.fit(X_train, y_train)
best_param = grid_search.best_params_['C']

# apply the best model to the test set and calculate accuracy
best_model = grid_search.best_estimator_
acc_train = best_model.score(X_train, y_train)
acc_test = best_model.score(X_test, y_test)

Best C: 2
Accuracy on train set: 0.661
Accuracy on test set: 0.692
```

### 7.7.1.2 R

```
# Load necessary libraries
library(mlr3verse)
library(paradox) # for tuning
library(rsample) # for splitting data

df_movie_reviews_ = df_movie_reviews %>%
  mutate(rating_good = as.factor(rating_good)) |>
  select(matches('sc|rating_good'))

# split the dataset into training and test sets

splits = initial_split(df_movie_reviews_, prop = 0.75)

df_train = training(splits)
df_test = testing(splits)

# Define task
task = TaskClassif$new("movie_reviews", df_train, target = "rating_good")

# Define learner
learner = lrn("classif.glmnet", alpha = 0, predict_type = "response")

# Define resampling strategy
resampling = rsmp("cv", folds = 5)

# Define measure
measure = msr("classif.acc")
```

```

# Define parameter space
param_set = ParamSet$new(
  list(
    ParamDbl$new("lambda", lower = 1e-3, upper = 1)
  )
)

# Define tuner
tuner = AutoTuner$new(
  learner = learner,
  resampling = resampling,
  measure = measure,
  search_space = param_set,
  tuner = tnr("grid_search", resolution = 10),
  terminator = trm("evals", n_evals = 10)
)

# Tune hyperparameters
tuner$train(task)

# Get best hyperparameters
best_param = tuner$model$learner$param_set$values

# Use the best model to predict and get metrics
acc_train = tuner$predict(task)$score(msr("classif.acc"))
acc_test = tuner$predict_newdata(df_test)$score(msr("classif.acc"))

Best lambda: 0.445
Accuracy on train set: 0.6826666666666667
Accuracy on test set: 0.688

```

So there you have it. We searched a parameter space, chose the best set of parameters via k-fold cross validation, and got an assessment of generalization error in just a couple lines of code. Neat!

### 7.7.1.3 Search Spaces

In the previous example, we used a grid search to search over a range of values for the penalty parameter. This is a very simple approach, but it can be computationally expensive. We can do better by using a more sophisticated approach to search over the parameter space. For example, we can use a **random search**, where we randomly sample from the parameter space. This is generally faster than a grid search, and can be just as effective. Other methods are available that better explore the space and do so more efficiently.

💡 Grid search can work to some extent and is a quick and easy way to get started, but generally we want something that can search a true space rather than a limited grid. Typical options are random, bayesian optimization, hyperband, and genetic algorithms. Most of these are available in `scikit-learn` and `m1r3`.

## 7.8 Pipelines

For **production-level** work, or just for **reproducibility**, it is often useful to create a **pipeline** for your modeling work. A pipeline is a series of steps that are performed in sequence. For example, we might want to perform the following steps:

- Impute missing values
- Transform features
- Create new features
- Split the data into training and test sets
- Fit the model on the training set
- Assess the model's performance on the test set
- Compare the model with others
- Save the 'best' model
- Use the model for prediction on future data, sometimes called **scoring**
- Redo the whole thing from time to time

We can create a pipeline that performs all of these steps in sequence. This is useful for a number of reasons. First, doing so makes it far easier to reproduce the results as needed. Second, it is relatively easy to change the steps in the pipeline. For example, we might want to try a different imputation method, or add a new model. Third, it is relatively easy to apply the pipeline. For example, we might want to use the model on new data. We can just apply the pipeline to the new data, and it will perform all of the steps in sequence, including fitting the model. Fourth, having a pipeline facilitates model comparison, as we can ensure that the models are receiving the same data process. Finally, we can save the pipeline for later use- we just save the pipeline as a file, and then load it later when we want to use it again.

### 7.8.0.1 Python

Here is an example of a pipeline in Python. We use the `make_pipeline` function from the `sklearn` package. This function takes a series of steps as arguments, and then performs them in sequence. We can then use the pipeline to fit the model, assess its performance, and save it for later use.

```

# import necessary libraries
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegressionCV
from sklearn.metrics import accuracy_score

# create pipeline
pipeline = make_pipeline(
    SimpleImputer(strategy='mean'),
    StandardScaler(),
    LogisticRegressionCV(penalty='l2', Cs=[1], cv=5, max_iter=1000),
)

# fit the pipeline
pipeline.fit(X_train, y_train)

# assess the pipeline
y_pred = pipeline.predict(X_test)
accuracy_score(y_test, y_pred)

# save the pipeline
# from joblib import dump, load
# dump(pipeline, 'pipeline.joblib')

```

### 7.8.0.2 R

With R, `mlr3` works in a very similar fashion to `sklearn`, which is why we use it for demonstration. We create a pipeline with the `po`, or pipe operator function, which takes a series of steps as arguments, and then performs them in sequence.

```

# Load necessary libraries
library(mlr3verse)
# library(mlr3learners)
# library(mlr3pipelines)

# Define task
task = TaskClassif$new("movie_reviews", df_movie_reviews, target = "rating_good")

# Define learner
learner = lrn("classif.cv_glmnet", predict_type = "response")

# Define pipeline
pipeline = po("scale") %>>%
  po("imputemean") %>>%

```

```

po("learner", learner)

# Fit pipeline
pipeline$train(task)

# Assess pipeline
pipeline$predict(task)[[1]]$score(msr("classif.acc"))

# Save pipeline
# saveRDS(pipeline, "pipeline.rds")

```

Development and deployment of pipelines will depend on your specific use case, and can get notably complicated. Think of your model data being the culmination of features drawn from dozens of wildly different databases, and the model itself being a complex ensemble of models, each with their own hyperparameters. You can imagine the complexity of the pipeline that would be required to handle all of that, but it is possible. In any event, the basic idea is the same, and pipelines are a great way to organize your modeling work.

---

## 7.9 Commentary

When machine learning began to take off, it seemed many in the field of statistics sat on their laurels, and often scoffed at these techniques that didn't bother to test their assumptions<sup>13</sup>! ML was, after all, mostly just a rehash of old ideas right? But the machine learning community, which actually comprised both computer scientists and statisticians, was able to make great strides in predictive performance, and the application of machine learning in myriad domains continues to enable us to push the boundaries of what is possible. Statistical analysis wasn't going to provide ChatGPT or self-driving cars, but it remains vitally important whenever we need to understand the uncertainty of our predictions, or when we need to make inferences about the data world. A more general field of **data science** became the way people used statistics *and* machine learning to solve their data challenges. So the two fields are complementary and overlapping, and the best data scientists will be able to draw from both. In the end, use the best tool for the job, worry less about what

---

<sup>13</sup>To paraphrase provocatively, ‘machine learning is statistics minus any checking of models and assumptions’. Brian D. Ripley useR! 2004, Vienna (May 2004) Want to know what’s even crazier than that statement? It was said by the guy that literally wrote the book on neural networks<sup>14</sup> before anyone was even using them in any practical way! Also interesting to note is that techniques like random forests and others associated with machine learning actually came from established statisticians. In short, there never was a statistics vs. machine learning divide. Tools are tools, and the best data scientists will have many at their disposal for any project.

it's called or whether it's the hot thing, or has a cool label, and importantly, just have fun!

MOVE TO APPENDIX OR Part 3

---

## 7.10 Using R and Python in ML

### 7.10.1 Python

Python is the king of ML. Many other languages can perform ML and maybe even well, but Python is the most popular, and has the most packages, and it's where tools are typically implemented and developed first. Even if it isn't your primary language, it should be for any implementation of machine learning.

Pros:

- powerful and widely used tools
- typically very efficient on memory and fast
- many modeling packages try to use the sklearn API for consistency<sup>15</sup>
- easy pipeline/reproducibility setup

Cons:

- Everything beyond getting a prediction can be difficult: e.g. good model summaries and visualizations, interpretability tools, extracting key estimated model features, etc. For example, getting feature names as part of the output is a recent development for scikit-learn and other modeling packages.
- Data processing beyond applying simple functions to columns can be notably tedious. Pandas, to put it simply, is not tidyverse.
- The ML ecosystem is fragile, and one package's update will often break another package's functionality, meaning your work will often be frozen in time to whenever you first began model exploration. Many corporate modeling environments are still based on versions of Python that may be many years old, and the model packages will contain all the bugs from the time of release when the Python environment was created.
- Package documentation is often quite poor, even for some important model aspects of the model, and there is no consistency from one package to another. Demos may work or not, and you may have to dig into the source code to figure out what's actually going on. This hopefully will be alleviated in the future with modern AI tools that can write the documentation for you.
- Interactive model development with Jupyter has not been close to the level

---

<sup>15</sup>Note to developers, just having a fit and predict method is not an API. But as scikit-learn is not internally consistent in using its own API, it's not surprising that other packages don't either.

with alternatives like RMarkdown for years. However, Quarto has already shown great promise, as this book was written with it, so in the end, the R folks may bail out this issue for the Python folks.

### 7.10.2 R

Speaking as folks who've used tools like mlr3, tidyverse, and more on millions of data points for very large and well-known companies, we can say definitively that R is actually great at ML and at production level. The tools are not as fast or memory efficient relative to Python, but they are typically more user friendly, and usually have good to even excellent documentation, as package development has been largely standardized for some time. As far as some drawbacks, some Python packages such as xgboost and lightgbm have concurrent development in R, but even then the R development typically lags with feature implementation. And when it comes to ML with deep learning models, R packages merely wrap the underlying Python packages. In general though, for everything before and after ML, from feature engineering to visualization to reporting, R has much more to offer.

Pros:

- very user friendly and fast data processing
- easy to use objects that contain the things you'd need to use for further processing
- practically every tool you'd use works with data frames
- saving models does not require any special effort
- easy post-processing of models with many packages designed to work with the output of other modeling packages (e.g. broom, tidybayes, etc.)
- documentation is standardized for any CRAN and most non-CRAN packages, and will only improve with AI tools. Unlike Python, examples are expected for documented functions, and the package will fail to build if *any* example fails, and warn if examples are empty. This is a great way to ensure that examples are present and actually work.
- ML tools can be used on tabular data of millions of instances in memory and in production, and on data that is too large to fit in memory using disk-backed data structures.

Cons:

- relatively slow
- memory intensive
- pipeline/reproducibility has only recently been of focus
  - `tidymodels` is a great but fairly non-standard way of conducting machine learning
  - `mlr3` is much more `sklearn`-like- fast and memory efficient, but not as widely used
- developers often don't do enough testing

In summary, Python is the best tool for ML, but you can use R for pretty much everything else if you want, including ML if it's not too computationally expensive or you don't have to worry about that aspect. Quarto makes it easy to use both, including simultaneously, so the great thing is you don't have to choose!

---

## 7.11 Where to go from here

### 7.11.1 refs

ESL for R/Python

ridge as Bayesian WIKILINK: [https://en.wikipedia.org/wiki/Ridge\\_regression#Bayesian\\_interpretation](https://en.wikipedia.org/wiki/Ridge_regression#Bayesian_interpretation)  
dropout [https://d2l.ai/chapter\\_multilayer-perceptrons/dropout.html](https://d2l.ai/chapter_multilayer-perceptrons/dropout.html)

bv tradeoff

<https://hastie.su.domains/Papers/ESLII.pdf>

<https://machinelearningmastery.com/gentle-introduction-to-the-bias-variance-trade-off-in-machine-learning/>

RF/boosting <https://developers.google.com/machine-learning/decision-forests>

“Reconciling modern machine-learning practice and the classical bias–variance trade-off”, 2019, by Belkin, Hsu, Ma, Mandal, <https://www.pnas.org/doi/10.1073/pnas.1903070116>.

CV [https://scikit-learn.org/stable/auto\\_examples/model\\_selection/plot\\_nested\\_cross\\_validation\\_iris.html](https://scikit-learn.org/stable/auto_examples/model_selection/plot_nested_cross_validation_iris.html)

DL

Annotated History of Modern AI and Deep Learning, Juergen Schmidhuber Interpretation

Molnar

Techniques to Improve Ecological Interpretability of Black-Box Machine Learning Models <https://link.springer.com/article/10.1007/s13253-021-00479-7>



# 8

---

## *Common Models*

---

Let's get one thing straight from the outset: **any model may be used in machine learning**, from a standard linear model to a deep neural network. The key focus in ML is on performance, and generally we'll go with what works. This means that the modeler is often less concerned with the interpretation of the model, but rather with the ability of the model to predict well on new data, but as we'll see we can do both if desired. In this chapter, we will explore some of the more common machine learning models and techniques.

---

### 8.1 Key Ideas

The take home messages from this section include the following:

- Any model can be used with machine learning
- A good and simple baseline is essential for interpreting your performance results
- One only needs a small set of tools (models) to go very far with machine learning

#### 8.1.1 Why this matters

Having good choices in your data science toolbox means you don't have to waste time with nuance and can get down to what matters- performance! Furthermore, using these common tools means you'll know you're in good company, and that you'll be able to find many resources to help you along the way. Additionally, you'll be able to focus on the data and the problem at hand, rather than the model, which in the end, is just a tool to help you understand the data. If you can get a good understanding of the data with a simple model, then that may be all you need for your situation. If you decide you need a more complex modeling approach, then using these models will still give you a good idea of what you should expect in terms of performance.

### 8.1.2 Good to know

TODO: ADD LINK TO ESTIMATION CHAPTER

Before diving in, it'd be helpful to be familiar with the following:

- Linear models, esp. linear and logistic regression
  - Basic machine learning concepts as outlined in the the ML Concepts chapter (Chapter 7)
  - Model estimation as outlined in the Estimation chapter (Chapter 4)
- 

## 8.2 General Approach

Let's start with a general approach to machine learning to help us get some bearings. Here is an example outline of the process we could take. This incorporates some of the ideas we've already discussed, and we'll demonstrate most of this in the following sections.

- Define the problem, including the target variable(s)
- Select the model(s) to be used, including one baseline model
- Define the performance objective and metric(s) used for model assessment
- Define the search space (parameters, hyperparameters) for those models
- Define the search method (optimization)
- Implement some sort of cross-validation technique and collect the corresponding performance metrics
- Evaluate the results on unseen data with the chosen model
- Interpret the results

Here is a more concrete example:

- Define the problem: predict the probability of heart disease given a set of features
- Select the model(s) to be used: ridge regression, standard regression with no penalty as baseline
- Define the objective and performance metric(s): (R)MSE, R-squared
- Define the search space (parameters, hyperparameters) for those models: penalty parameter
- Define the search method (optimization): grid search
- Implement some sort of cross-validation technique: 5-fold cross-validation
- Evaluate the results on unseen data: RMSE on test data
- Interpret the results: the ridge regression model performed better than the baseline model, and the coefficients tell us something about the nature of the relationship between the features and the target

As we go along in this chapter, we'll most of this in action at various points.

We'll have a baseline model, an ultimately provide examples of several commonly used models in machine learning. In each case we will assess performance using cross-validation, and then evaluate the final models on unseen data. Separately, we'll also demonstrate how to tune hyperparameters, which are parameters that are not estimated directly from the data, but rather are set by the modeler. We'

---

### 8.3 Data setup

FIXME: add appendix link for dataset! [LINK](#) to DATA Sections!

For our demonstration here, we'll switch things up and use the heart disease dataset. This is a binary classification problem, where we want to predict whether a patient has heart disease, given information such as age, sex, resting heart rate etc. For more details see the appendix. We have done some initial data processing so that you can dive right in.

*There are two forms of the data - one which is mostly as seen elsewhere, and one that is purely numeric, where the categorical features are dummy coded and where numeric variables have been standardized (Section 10.2). The purely numeric version will save any additional data processing for some model/package implementations. We also have to drop missing values, so that our 21 century packages don't hurt themselves on them. When we get to the boosting demonstration, you can use the data as is, since the scale of the data doesn't really matter, and missing values are treated in the same way as the rest of the data values.*

In this data, roughly 54% suffered a death, so that is an initial baseline if we're interested in accuracy- we could get 46% correct by just guessing the majority class.

#### 8.3.0.1 Python

```
import pandas as pd
import numpy as np

df_heart = pd.read_csv('data/heart_disease_processed.csv')
df_heart_num = pd.read_csv('data/heart_disease_processed_numeric_sc.csv')

# convert appropriate features to categorical
for col in df_heart.select_dtypes(include='object').columns:
    df_heart[col] = df_heart[col].astype('category')

X = df_heart_num.drop(columns=['heart_disease']).to_numpy()
```

```
y = df_heart_num['heart_disease'].to_numpy()

# some models can't automatically handle missing data
y_complete = df_heart_num.dropna()['heart_disease'].to_numpy().astype(int)
X_complete = df_heart_num.dropna().drop(columns='heart_disease').to_numpy()
```

### 8.3.0.2 R

```
library(tidyverse)

df_heart = read_csv("data/heart_disease_processed.csv") |>
  mutate(across(where(is.character), as.factor))

df_heart_num = read_csv("data/heart_disease_processed_numeric_sc.csv")

# as a data.frame for mlr3
X_num_df = df_heart_num %>%
  as_tibble() |>
  mutate(heart_disease = factor(heart_disease)) |>
  janitor::clean_names() # remove some symbols
```

---

## 8.4 Do Better than the Baseline

CAN WE GET A VISUAL IN HERE SOMEWHERE?

Before getting carried away with models, we should try and get something that gives us a good reference point for performance - a **baseline** model. The baseline model should serve as a way to gauge how much better your model performs over one that is simpler, probably more computationally efficient, and more interpretable. Or maybe it's one that is sufficiently complex to capture something about the data you are exploring, but not as complex as the models you're also interested in. Take a classification model for example. We use a logistic regression as baseline, which is as simple as it gets, but is often too simple to be adequately performant for many situations. Even so, we should still be able to beat it with more complex models, or there is little justification for using them.

### 8.4.1 Why do we do this?

You can actually find articles in which deep learning models do not even beat a logistic regression on some datasets, but the fact of which did not stop the authors writing several pages hyping the more complex technique. Probably

the most important reason to have a baseline is so that you can avoid wasting time and resources implementing more complex tools, or simply getting excited for no good reason. It is probably rare, but sometimes relationships for the chosen features and target are mostly or nearly linear and have little interaction, and no amount of fancy modeling will make it come about. Furthermore, if our baseline is a complex linear model that actually incorporates nonlinear relationships and interactions (e.g. a GAMM), you'll often find that the more complex models don't significantly improve on the baseline by much, if at all. In addition, in time series settings, a moving average or last target value can often be a very good predictor. So in general, you may find that the initial baseline model is good enough for the time being, and you can then move on to other problems to solve, like acquiring data that is functionally predictive. This is especially true if you are working in a business setting where you have limited time and resources.

A final note. In many (most?) settings, it often isn't enough to merely beat the baseline model. You should look to do statistically better. For example, if your complex model accuracy is 75% and your baseline is 73%, that's great, but you should check to see if that difference is statistically significant<sup>1</sup>, because those metrics are *estimates*, and they have uncertainty, which means you can get a range for them as well as test whether they are different from one another. If the difference is not notable, then you should probably stick with the baseline model or try something else, because the next time you run the model, the baseline may actually perform better, or at least you can't be sure that it won't.

That said, in some situations *any* performance increase is worth it, and even if we can't be certain a result is statistically better, any sign of improvement is worth pursuing. For example, if you are trying to predict the next word in a sentence, and your baseline is 10% accurate, and your complex model is 11% accurate, that's a 10% increase in accuracy, which may be a big deal for user experience. You should still work to show that this is a consistent increase and not a fluke.

---

## 8.5 Penalized Linear Models

TODO: ADD LINK TO ESTIMATION CHAPTER

<sup>1</sup>There would be far less hype and wasted time if those in ML and DL research simply did this rather than just reporting the chosen metric of their model 'winning' against other models. It's not that hard to do, yet most do not provide any ranged estimate for their metric, let alone test statistical difference from other models. You don't even have to bootstrap the metric estimates for binary classification! It'd also be nice if they used a more meaningful baseline than logistic regression, but that's a different story.

So let's get on with some models already! Let's use the classic linear model as our starting point for ML, just because we can. We show explicitly how to estimate models like lasso and ridge regression in Section 4.9. Those work well as a baseline, and so should be in your ML toolbox.

### 8.5.1 Elastic Net

Another common linear model approach is **elastic net**, which is a combination of lasso and ridge. We will not show how to estimate elastic net by hand here, but all you have to know is that it combines two penalties, the same ones for lasso and one for ridge, along with the standard objective for a numeric or categorical target. The relative size of the two penalties is controlled by a mixing parameter, and the optimal value of that parameter is determined by cross-validation. So for example, you might end up with a 75% lasso penalty and 25% ridge penalty. In the end though, we're just going to do a slightly fancier logistic regression!

Let's apply this to the heart disease data. We'll used the 'processed version' which has dummy codes and has dropped the few observations with missing values. We are only doing simple cross-validation here to get a better performance assessment, but you are more than welcome to tune both the penalty parameter and the mixing ratio as we have demonstrated before. We'll revisit hyperparameter tuning towards the end of this chapter.

#### 8.5.1.1 Python

```
from sklearn.linear_model import LogisticRegression, LogisticRegressionCV
from sklearn.model_selection import cross_validate, KFold, cross_val_score
from sklearn.metrics import accuracy_score

model_elastic = LogisticRegression(
    penalty='elasticnet',
    solver='saga',
    l1_ratio=0.5,
    random_state=42,
    max_iter=10000,
    verbose=False,
)

# model_elastic.fit(X_complete, y_complete)

# use cross-validation to estimate performance
cv_elastic = cross_validate(
    model_elastic,
    X_complete,
```

```

    y_complete,
    cv=5,
    scoring='accuracy',
)

# pd.DataFrame(cv_elastic) # default output

Training accuracy: 0.829
Baseline: 0.541

```

### 8.5.1.2 R

```

library(mlr3verse)

tsk_elastic = as_task_classif(
  X_num_df |> drop_na(),
  target = "heart_disease"
)

lrn_elastic = lrn(
  "classif.cv_glmnet",
  nfolds = 5,
  type.measure = "class",
  alpha = 0.5
)

cv_elastic = resample(
  task      = tsk_elastic,
  learner   = lrn_elastic,
  resampling = rsmp("cv", folds = 5)
)

# cv_elastic$aggregate(msr('classif.acc')) # default output

Training Accuracy: 0.839
Baseline Prevalence: 0.541

```

So we're starting off with what seems to be a good model. Our average accuracy across the validation sets is definitely doing better than guessing, an increase of almost 55%! Now let's see if we can do better with other models!

### 8.5.2 Strengths & Weaknesses

#### Strengths

- Intuitive approach. In the end, it's still just a standard regression model you're already familiar with.

- Widely used for many problems. Lasso/Ridge/ElasticNet would be fine to use in any setting you would use linear or logistic regression.

### Weaknesses

- Does not automatically seek out interactions and non-linearity, and as such will generally not be as predictive as other techniques.
- Variables have to be scaled or results will largely reflect data types.
- May have issues with correlated predictors

### 8.5.3 Additional Thoughts

Incorporating regularization as done with penalized regression would be fine as your default linear model method, and is something to strongly consider for even statistical model settings. Furthermore, these approaches will have better prediction on new data than their standard, nonregularized complements. As such they are a nice balance between staying interpretable while enhancing predictive capability. However, in general they are not going to be as strong of a method as others in the ML universe, and possibly not even competitive without a lot of feature engineering. If prediction is all you care about for a particular modeling setting, you'll likely want to try something else.

## 8.6 Tree-based methods

Let's move beyond standard linear models and get into a notably different type of approach. Tree-based methods are a class of models that are very popular in machine learning, and for good reason, they work *very well*. To get a sense of how they are derived, consider the following classification example where we want to predict a binary target as 'Yes' or 'No'. We have two numeric features,  $X_1$  and  $X_2$ . At the start we take  $X_1$  and make a split at the value of 5. Any observation less than 5 on  $X_1$  goes to the right with a prediction of *No*. Any observation greater than or equal to 5 goes to the left, where we then split based on values of  $X_2$ , and specifically at 3. Any observation less than 3 goes to the right with a prediction of *Yes*. Any observation greater than or equal to 3 (and greater than or equal to 5 on  $X_1$ ) goes to the left with a prediction of *No*. So in the end, we see relatively lower on  $X_1$ , or relatively higher on both, results in a prediction of *No*, and high on  $X_1$  and low on  $X_2$  results in a prediction of *Yes*. We can see this visually in the following graph.

This is a simple example, but it illustrates the basic idea of a tree-based model, where the **tree** reflects the total process, and **branches** are represented by the splits going down, ultimately ending at **leaves** where predictions are made. We can also think of the tree as a series of **if-then** statements, where we start

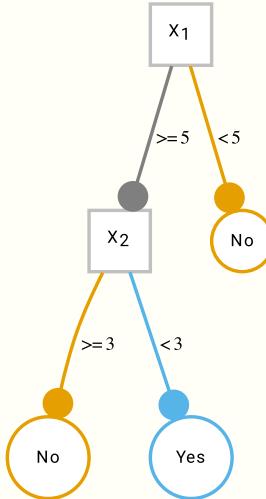


Figure 8.1: A simple classification tree

at the top and work our way down until we reach a leaf node, which is a prediction for all observations that qualify for that leaf.

If we just use a single tree, this would be the most interpretable model we could probably come up with, and it incorporates nonlinearities (multiple branches on a single feature), interactions (branches across features), and feature selection all in one (some features may not result in useful splits for the objective). However, a single tree is not a very stable model unfortunately, and so does not generalize well. For example, just a slight change in data, or even just starting with a different feature, might produce a very different tree<sup>2</sup>. The solution is straightforward though - by using the power of a bunch of trees, we can get predictions for each observation from each tree, and then average the predictions, result in a most stable estimate. This is the concept behind both **random forests** and **gradient boosting**, which can be seen as different algorithms to produce a bunch of trees, and then average the predictions. They also fall under the heading of **ensemble models**, which are models that combine the predictions of multiple models, in this case individual trees, to ultimately produce a single prediction for each observation.

Random forests and boosting methods are very easy to implement, to a point. However, there are typically a several hyperparameters to consider for tuning. Here are just a few to think about:

- Number of trees

---

<sup>2</sup>A single regression/classification tree actually could serve as a decent baseline model, especially given the interepretability.

- Learning rate (GB)
- Maximum **depth** of each tree
- Minimum number of observations in each leaf
- Number of features to consider at each tree/split
- Regularization parameters (GB)
- Out-of-bag sample size (RF)

Those are the ones that you'll usually be trying to figure out via cross-validation for boosting or random forests, but there are others. The number of trees and learning rate kind of play off of each other, where having more trees allows for a smaller rate<sup>3</sup>, which might work better but will take longer to train, and can lead to overfitting if other steps are not taken. The depth of each tree refers to the number of levels down the branches we allow the model to go, as well as how wide we let things get in some implementations. This is important because it controls the complexity of each tree, and thus the complexity of the overall model- less depth helps to avoid overfitting, but too little depth and you won't be able to capture the nuances of the data. The minimum number of observations in each leaf is also important for the same reason. It's also generally a good idea to take a random sample of features for each tree (or possibly even each branch), to also help reduce overfitting, but it's not obvious what proportion to take. The regularization parameters are typically less important in practice, but in general you can use them to reduce overfitting as we would in other modeling circumstances.

Here is an example of gradient boosting with the heart disease data. Although boosting methods are available in `scikit-learn` for Python, in general we recommend using `lightgbm` or `xgboost` packages directly for boosting implementation, which have a `sklearn` API anyway (as demonstrated). Also, they both provide R and Python implementations of the package, making it easy to not lose your place when switching between languages. We'll use `lightgbm` here, but `xgboost` is also a very good option <sup>4</sup>.

#### 8.6.0.1 Python

```
# potential models you might use
from sklearn.ensemble import HistGradientBoostingClassifier
from lightgbm import LGBMClassifier
from xgboost import XGBClassifier, DMatrix

from sklearn.metrics import accuracy_score
```

---

<sup>3</sup>This is pretty much the same concept as in stochastic gradient boosting. Larger learning rates allow for quicker exploration, but may overshoot the optimal value, however defined. Smaller learning rates are more conservative, but may take longer to find the optimal value.

<sup>4</sup>Some also prefer `catboost`. The authors have not actually been able to practically implement catboost in a setting where it was more predictive or as efficient/speedy as `xgboost` or `lightgbm`, but some have had notable success with it.

```

model_boost = LGBMClassifier(
    n_estimators=1000,
    learning_rate=1e-3,
    max_depth = 5,
    verbose = -1
)

cv_boost = cross_validate(
    model_boost,
    df_heart.drop(columns='heart_disease'),
    df_heart_num['heart_disease'],
    cv=5,
    scoring='accuracy',
)

```

Training accuracy: 0.838  
Baseline Prevalence: 0.541

### 8.6.0.2 R

Note that as of writing, the mlr3 implementation of lightgbm doesn't seem to handle factors even though the R package does. So we'll use the numeric version of the data here.

```

library(mlr3verse)
# for lightgbm, you need mlr3extralearners and lightgbm package installed
# remotes::install_github("mlr-org/mlr3extralearners@*release")
library(mlr3extralearners)

set.seed(1234)

# Define task
# For consistency we use X_num_df, but lgbm can handle factors and missing data
# and so we can use the original df_heart if desired
tsk_boost = as_task_classif(
    X_num_df,
    target = "heart_disease"
)

# Define learner
learner_boost = lrn(
    "classif.lightgbm",
    num_iterations = 1000,
    max_depth = 5,
    learning_rate = 1e-3
)

```

```
# Cross-validation
cv_boost = resample(
  task      = tsk_boost,
  learner   = learner_boost,
  resampling = rsmp("cv", folds = 5)
)

Training Accuracy: 0.828
Baseline Prevalence: 0.541
```

So here we have a model that is also performing well, though not significantly better or worse than our elastic net model. For most situations, we'd expect boosting to do better, but this shows why we want a good baseline or simpler model. We'll revisit hyperparameter tuning using this model later. If you'd like to see an example of how we could implement a form of gradient boosting by hand<sup>5</sup>, see the appendix.

#### ADD GBLINEAR BY HAND TO APPENDIX

##### 8.6.0.3 Strengths & Weaknesses

Random forests and boosting methods, though not new, are still ‘state of the art’ in terms of performance on tabular data like the type we’ve been using for our demos here. As of this writing, you’ll find that it will usually take considerable effort to beat them on tabular data.

#### Strengths

- A single tree is highly interpretable.
- Easily incorporates features of different types (the scale of numeric features, or using categoricals, doesn’t matter).
- Tolerance to irrelevant features.
- Some tolerance to correlated inputs.
- Handling of missing values. Missing values are just another value to potentially split on.

#### Weaknesses

- Honestly few, but like all techniques, it might be relatively less predictive in certain situations. There is no free lunch<sup>6</sup>.
- It does take more effort to tune relative to linear model methods.

---

<sup>5</sup>LINKYLINK

<sup>6</sup><https://machinelearningmastery.com/no-free-lunch-theorem-for-machine-learning/>

## 8.7 Deep Learning and Neural Networks

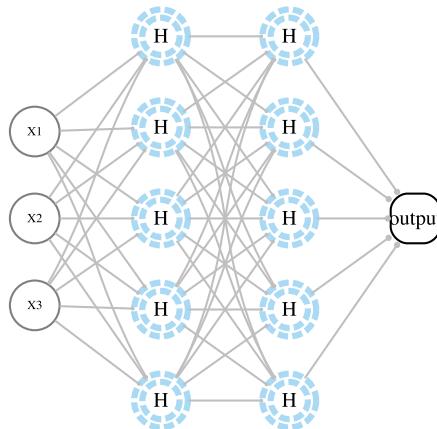


Figure 8.2: A neural network

**Deep learning has fundamentally transformed the world of data science.** It has been used to solve problems in image recognition, speech recognition, natural language processing, and more, from assisting with cancer diagnosis to summarizing entire novels. Deep learning has also been used to solve problems with tabular data of the kind we've been focusing on. As yet, it is not a panacea for every problem, and is not always the best tool for the job, but it is a tool that should be in your toolbox. Here we'll provide brief overview of the key concepts behind neural networks, the underlying technology behind deep learning, and then demonstrate how to implement a simple neural network to get things started.

### 8.7.1 What is a neural network?

Neural networks have actually been around a while. Computationally, since the 80s, and conceptually even much further back. They were not very popular for a long time, but this was mostly a computing limitation, much the same reason Bayesian methods were slower to develop relative to related alternatives. But now neural networks have recently become the go-to method for many problems. They still can be very computationally expensive, but we at least have the hardware to pull it off now.

At its core, a neural network can be seen as complex series of matrix multiplications exactly as we've done with a basic linear model. One notable difference is that neural networks actually implement multiple combinations of features (often referred to as hidden **nodes** or **units**), and we add in nonlin-

ear transformations between the matrix multiplications, typically referred to as **activations**. In fact, you can actually think of neural networks as nonlinear extensions of linear models<sup>7</sup>. The linear part is just like a standard linear model, where we have a set of features, each with a corresponding weight, and we multiply each feature by its weight and sum them up. The activation part is where things start to get more interesting, where we take the output of the linear part and apply a transformation to it, allowing the model to incorporate nonlinearities. Furthermore, by combining multiple linear parts and activations together, then repeating the whole process for yet another **layer** of the model but using the hidden nodes as inputs for the subsequent combinations, we can incorporate interactions between features.

Before getting carried away, let's simplify things a bit. We have multiple options for our activation functions, the most common one being what's called the **rectified linear unit** or ReLU<sup>9</sup>. But, we could also use the sigmoid function<sup>10</sup>, which is exactly the same as the logistic link function used in logistic regression. In logistic regression, we take the linear combination of features and weights, and then apply the sigmoid function to it. Because of this, we can actually think of logistic regression as a very simple neural network, with a the linear combination as a single hidden node and a sigmoid activation function adding the nonlinear transformation!

The following shows a logistic regression as a neural network. The input features are  $X_1$ ,  $X_2$ , and  $X_3$ , and the output is the probability of a positive outcome of a binary target. The weights are  $w_1$ ,  $w_2$ , and  $w_3$ , and the bias<sup>11</sup> is  $w_0$ . The hidden node is just our linear predictor which we can create via matrix multiplication of the input matrix and weights. The sigmoid function is the activation function, and the output is the probability of the chosen label.

### 8.7.2 Trying it out

TODO: ADD LINK TO DATA CHAPTER re EMBEDDINGS

For simplicity we'll use the same approach and tools as before, but do know this is probably the very bare minimum approach for a neural network, and generally you'd prefer an alternative. Our model is a **multi-layer perceptron** (MLP), which consists of multiple hidden layers of varying sizes. To begin with, you'd likely want to tune the architecture a bit in normal circumstances

---

<sup>7</sup>Regression approaches like GAMs and gaussian process regression can be seen as approximations to neural networks<sup>8</sup>. This brings us back to having a good baseline. If you know some simpler tools that can approximate more complex ones, you can often get 'good enough' results with the simpler models.

<sup>9</sup>[https://en.wikipedia.org/wiki/Rectifier\\_\(neural\\_networks\)](https://en.wikipedia.org/wiki/Rectifier_(neural_networks))

<sup>10</sup>[https://en.wikipedia.org/wiki/Sigmoid\\_function](https://en.wikipedia.org/wiki/Sigmoid_function)

<sup>11</sup>It's not exactly clear why computer scientists chose to call this the bias<sup>12</sup>, but it's the same as the intercept in a linear model, or conceptually as an offset or constant. It has nothing to do with the word bias as used in every other modeling context.

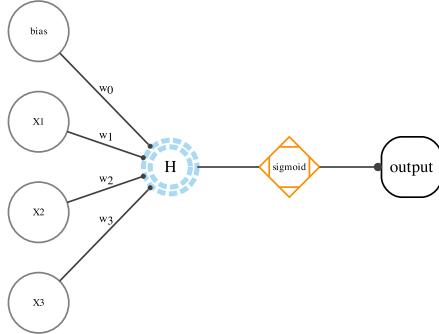


Figure 8.3: A logistic regression as a neural network

just as a starting point. Also, as noted in the data discussion, we'd usually want to use **embeddings** for categorical features as opposed to the one-hot approach used here, although it amounts to much the same thing, just with some additional computational load<sup>13</sup>.

For our example, we'll use the processed data with one-hot encoded features. For our architecture, we'll use three hidden layers with 200 nodes each. As noted, these and other settings are hyperparameters that you'd normally prefer to tune.

#### 8.7.2.1 Python

For our demonstration we'll use `sklearn`'s builtin `MLPClassifier`. We set the learning rate to 0.001. We'll also use a validation set of 20% of the data to help with early stopping. We set an **adaptive learning rate**, which is a way to automatically adjust the learning rate as the model trains. The `relu` activation function is default. We'll also use the **nesterov momentum** approach, which is a way to help the model avoid local minima. We use a **warm start**, which allows us to train the model in stages, which is useful for early stopping. We'll also set the **validation fraction**, which is the proportion of data to use for the validation set. And finally, we'll use `shuffle` to randomly select observations for each batch.

```
from sklearn.neural_network import MLPClassifier

model_mlp = MLPClassifier(
    hidden_layer_sizes=(200, 200, 200),
    learning_rate='adaptive',
    learning_rate_init=0.001,
    shuffle=True,
```

---

<sup>13</sup>A really good tool for a standard MLP type approach with automatic categorical embeddings is fastai's tabular learner.

```

        random_state=123,
        warm_start=True,
        nesterovs_momentum=True,
        validation_fraction=.2,
        verbose=False,
    )

# with the above settings, this will take a few seconds
cv_mlp = cross_validate(
    model_mlp,
    X_complete,
    y_complete,
    cv=5
)

# pd.DataFrame(cv_mlp) # default output

Training accuracy: 0.829
Baseline Prevalence: 0.541

```

### 8.7.2.2 R

For R, we'll use `mlr3torch`, which calls `pytorch` directly under the hood. We'll use the same architecture as was done with the Python example. It uses the `relu` activation function as a defualt. We'll also use `adam` as the optimizer, which is a popular choice and the default for the `sklearn` approach also. We'll also use `cross entropy` as the loss function, which is the same as the log loss objective function used in logistic regression and other ML classification models. We use a `batch size` of 16, which is the number of observations to use for each batch of training<sup>14</sup>. We'll also use `epochs` of 200, which is the number of times to train on the entire dataset. We'll also use `predict type` of `prob`, which is the type of prediction to make. Finally, we'll use both `logloss` and `accuracy` as the metrics to track. As specified, this took over a minute.

```

library(mlr3torch)

learner_mlp = lrn(
  "classif.mlp",
  # defining network parameters
  layers = 3,
  d_hidden = 200,
  # training parameters
  batch_size = 16,
  epochs = 200,

```

---

<sup>14</sup><https://stats.stackexchange.com/questions/153531/what-is-batch-size-in-neural-network>

```

# Defining the optimizer, loss, and callbacks
optimizer = t_opt("adam", lr = 1e-3),
loss = t_loss("cross_entropy"),
# # Measures to track
measures_train = msrs(c("classif.logloss")),
measures_valid = msrs(c("classif.logloss", "classif.ce")),
# predict type (required by logloss)
predict_type = "prob",
seed = 123
)

tsk_mlp = as_task_classif(
  backend = X_num_df |> drop_na(),
  target = 'heart_disease'
)

# this will potentially take about a minute
cv_mlp = resample(
  task      = tsk_mlp,
  learner   = learner_mlp,
  resampling = rsmp("cv", folds = 5),
)

```

# cv\_mlp\$aggregate(msr("classif.acc")) # default output

Training Accuracy: 0.826  
 Baseline Prevalence: 0.541

This neural network model actually did pretty well, and we're on par with our accuracy as we were with the other two models. This is somewhat surprising given the nature of the data- small number of observations with different data types- a type of situation in which neural networks don't usually do as well as others. Just goes to show, you never know until you try!

### 8.7.2.3 Strengths & Weaknesses

#### Strengths

- Good prediction generally.
- Incorporates the predictive power of different combinations of inputs.
- Some tolerance to correlated inputs.

#### Weaknesses

- Susceptible to irrelevant features.
- Doesn't outperform other methods that are easier to implement on tabular data.

## 8.8 A Tuned Example

As we noted in the chapter on machine learning concepts, there are typically multiple hyperparameters we are concerned with. For the linear model, we might want to tune the penalty parameter and the mixing ratio and/or penalty value. For a boosting method, we might want to tune the number of trees, the learning rate, the maximum depth of each tree, the minimum number of observations in each leaf, and the number of features to consider at each tree/split. And for a neural network, we might want to tune the number of hidden layers, the number of nodes in each layer, the learning rate, the batch size, the number of epochs, and the activation function. And so on.

Here is an example using the boosted model from before. We'll use the same data and settings as before, but we'll tune the number of trees, the learning rate, and the maximum depth of each tree. We'll use a **randomized search** approach, which is a way to randomly sample from a set of hyperparameters, rather than searching every possible combination. This is a good approach when you have a lot of hyperparameters to tune, and/or when you have a lot of data.

### 8.8.0.1 Python

```
from sklearn.model_selection import RandomizedSearchCV, train_test_split
from sklearn.metrics import accuracy_score

from lightgbm import LGBMClassifier

# train-test split
X_train, X_test, y_train, y_test = train_test_split(
    df_heart.drop(columns='heart_disease'),
    df_heart_num['heart_disease'],
    test_size=0.2,
    random_state=42
)

model_boost = LGBMClassifier(
    verbose = -1
)

param_grid = {
    'n_estimators': [500, 1000],
    'learning_rate': [1e-3, 1e-2, 1e-1],
    'max_depth': [3, 5, 7, 9],
    'min_child_samples': [1, 5, 10],
```

```
}
```

```
# this will take a few seconds
cv_boost_tune = RandomizedSearchCV(
    model_boost,
    param_grid,
    n_iter = 10,
    cv=5,
    scoring='accuracy',
    n_jobs=-1
)

cv_boost_tune.fit(X_train, y_train)
```

```
Test Accuracy 0.82
Baseline Prevalence: 0.541
```

### 8.8.0.2 R

```
# train test split

set.seed(123)

library(mlr3verse)
library(rsample)

split = initial_split(df_heart, prop = .75)

df_train = training(split)
df_test = testing(split)

tsk_lgbm_tune = as_task_classif(
    df_train,
    target = "heart_disease"
)

lrn_lgbm_tune = lrn(
    "classif.lightgbm",
    num_iterations = to_tune(c(500, 1000)),
    learning_rate = to_tune(1e-3, 1e-1),
    max_depth = to_tune(c(2, 3, 5, 7, 9)),
    min_data_in_leaf = to_tune(c(1, 5, 10))
)

# set up the validation process
instance_lgbm_tune = ti(
```

```

task = tsk_lgbm_tune,
learner = lrn_lgbm_tune,
resampling = rsmp("cv", folds = 5),
measures = msr("classif.acc"),
terminator = trm("evals", n_evals = 10)
)

# instance
tuner = tnr("random_search")

tuner$optimize(instance_lgbm_tune)

Test Accuracy: 0.855
Baseline Prevalence: 0.541

```

---

## 8.9 Comparing models

Let's compare our models head to head. We went back and restarted our process. We first split the data into training and test sets, the latter a 25% holdout. Then with training, we tuned each model over different settings:

- Elastic net: penalty and mixing ratio
- Boosting: number of trees, learning rate, and maximum depth, etc.
- Neural network: number of hidden layers, number of nodes in each layer

After this, we used the tuned values to retrain on the complete data set. At this stage it's not necessary to investigate typically, but here we show the results of the 10-fold cross-validation for the already-tuned models, to give a sense of the uncertainty in error estimation.

When it came to the holdout set with our best models, we see something you might be surprised about - the simplest model wins! It means we can use the simpler model and not worry about the more complex one, or just that we'd be fine using whichever one we prefer. However, none of these results are likely *statistically different* from each other. As an example, the elastic net model had an accuracy of 0.88, but the interval estimate for such a small sample is very wide - from 0.78 to 0.94. The interval estimate for the *difference* in accuracy between the elastic net and boosting models is from -0.07 to 0.18<sup>15</sup>. This was a good example of the importance of having an adequate baseline, and where complexity didn't really help much, though all our approaches did well.

---

<sup>15</sup>We just used the `prop.test` function in R for these values with the test being, what proportion of predictions are correct, and are these proportions different? A lot of the metrics people look at from confusion matrices are proportions.

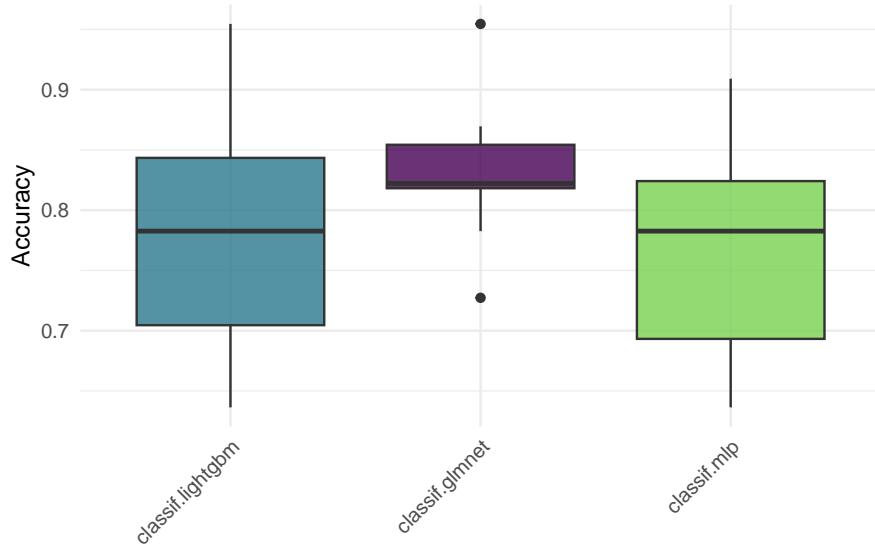


Figure 8.4: Cross-validation results for tuned models.

Table 8.1: Results for tuned models on holdout data.

model	Acc.	TPR	TNR	F1	PPV	NPV
Elastic Net	0.88	0.86	0.90	0.87	0.88	0.88
LGBM	0.83	0.80	0.85	0.81	0.82	0.83
MLP	0.84	0.77	0.90	0.82	0.87	0.82

**i** Some may wonder why the holdout results are better than the cross-validation results. This can happen, and at least in this case may mostly reflect the small sample size. The holdout set is a random sample of 20% of the complete data, 75 observations. Just a couple different predictions could result in a several percentage points difference in accuracy. Also, the holdout set is a random sample that is not the same data, so this could happen just by chance. In general though, you'd expect the holdout results to be a bit, or even significantly, worse than the cross-validation results, but not always.

## 8.10 Interpretation

When it comes to machine learning, just because we have some models at our disposal that don't readily lend themselves to interpretation with simple coefficients, it doesn't mean we can't still figure out what's going on. Let's use the boosting model as an example.

### 8.10.1 Feature Importance

The default importance metric for a lightgbm model is the number of splits in which a feature is used across trees, and this will depend notably on your settings and the chosen parameters of the best model. You could also use the Shap approach for variable importance as well, where importance is determined by average absolute Shap value. For this data and the model, depending on the settings, you might see that the most important features are age, cholesterol, and max heart rate.

#### 8.10.1.1 Python

```
# load the model
import joblib

cv_boost_tune = joblib.load('ml/data/tune-boost-py-model.pkl')

# Get feature importances
cv_boost_tune.feature_importances_
```

#### 8.10.1.2 R

R shows the proportion of splits in which a feature is used across trees rather than the raw number.

```
# load the tuned model
load("ml/data/tune-boost-r-results.RData")

# Get feature importances
lrn_lgbm_tuned$importance()
```

Feature	value
chest_pain_type_asymptomatic	0.28
num_major_vessels	0.16
thalassemia_normal	0.14
st_depression	0.09

Now let's think about a visual display. Here we demonstrate a quick partial dependence plot to see the effects of cholesterol and being male. We can see that males are expected to have a higher probability of heart disease, and that cholesterol has a positive relationship with heart disease, such that a notable rise begins around the mean value for cholesterol. The plot shown is a prettier version of what you'd get with the following code.

#### 8.10.1.3 Python

```
from sklearn.inspection import PartialDependenceDisplay

PartialDependenceDisplay.from_estimator(
    cv_boost_tune,
    df_heart.drop(columns='heart_disease'),
    features=['cholesterol', 'male'],
    categorical_features=['male'],
    percentiles=(0, .9),
    grid_resolution=75
)
```

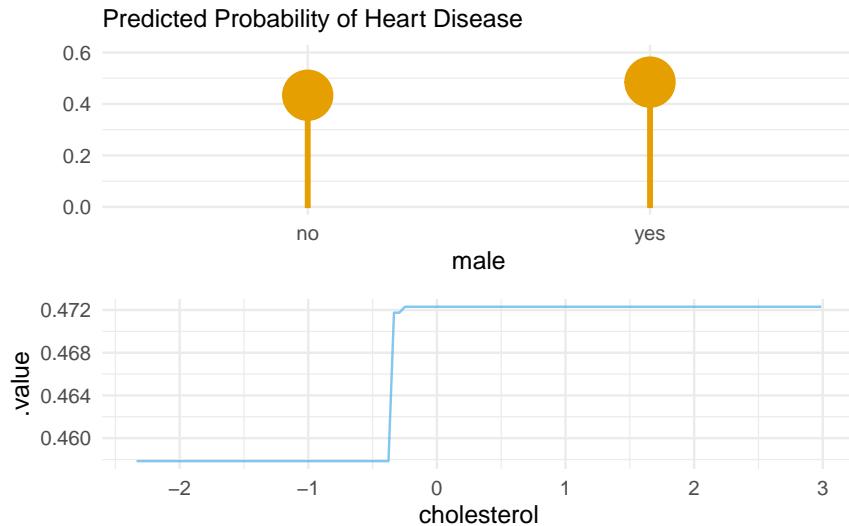
#### 8.10.1.4 R

For R we'll use the iml package.

```
library(iml)

prediction = Predictor$new(
    lrn_lgbm_tuned,
    data = df_train,
    type = 'prob',
    class = '1'
)

effect_dat = FeatureEffect$new(
    prediction,
    feature = c('cholesterol', 'male'),
    method = "pdp",
)
effect_dat$plot(show.data = TRUE)
```



## 8.11 Other ML Models for Tabular Data

When you look up models used in classical machine learning applied to data of the type we've been exploring, you'll potentially see a lot of different kinds. Popular methods from the past include  $k$ -nearest neighbors regression, support vector machines, and more. You don't see these used in practice much though, as these have mostly been made obsolete due to not being as predictive as other options in general ( $k$ -nn regression), making strong assumptions about the data distribution (linear discriminant analysis), maybe only works well with 'pretty' data situations (SVM), are computationally infeasible for larger datasets (most of them), or just being less interpretable.

While some of these models might still work well in unique situations, when you have tools that can handle a lot of data complexity and predict very well (and typically better) like tree-based methods, there's not much reason to use the historical alternatives these days. If you're interested in learning more about them or think one of them is just 'neat'<sup>16</sup>, you could potentially use it as a baseline model. Alternatively, you could maybe employ them as part of an ensemble model, where you combine the predictions of multiple models to produce a single prediction. This is a common approach in machine learning, and is often used in Kaggle competitions. We won't go into detail here, but it's worth looking into if you're interested. There are also many other methods

---

<sup>16</sup>Mathy folk should love SVMs.

that are more specialized, such as those for text, image, and audio data. We will provide an overview of these in another chapter.

---

## 8.12 Wrapping Up

In this chapter we've provided a few common and successful models you can implement with much success in machine learning. You don't really need much beyond these for tabular data unless your unique data condition somehow requires it. But a couple things are worth mentioning before moving on...

**Feature engineering will typically pay off more in performance than the model choice.**

**Thinking hard about the problem and the data is more important than the model choice.**

**The best model is simply the one that works best.**

You'll always get more payoff by coming up with better features to use in the model, as well as just using better data that's been 'fixed' because you've done some good exploratory data analysis. Thinking harder about the problem means you won't waste time going down dead ends, and you typically can find better data to use to solve the problem by thinking more clearly about the question at hand. And finally, it's good to not be stuck on one model, and be willing to use whatever it takes to get things done efficiently.

---

## 8.13 Exercise

Tune a model of your choice to predict whether a movie is good or bad with the movie review data<sup>17</sup>. Use the processed data which has the categorical outcome, and use one-hot encoded features if needed. Make sure you use a good baseline model for comparison!

---

<sup>17</sup>[data/movie\\_reviews\\_processed.csv](#)

---

## 8.14 Where to go from here

### 8.14.1 refs

RadfordM.Neal.Priorsforinfinitenetworks(tech.rep.no.crg-tr-94-1).UniversityofToronto, 1994a. <https://arxiv.org/abs/1711.00165>

[https://en.wikipedia.org/wiki/Activation\\_function](https://en.wikipedia.org/wiki/Activation_function)

<https://stats.stackexchange.com/questions/164876/what-is-the-trade-off-between-batch-size-and-number-of-iterations-to-train-a-neu>

# 9

---

## *More ML*

---

We've covered how to do some modeling for standard data settings and modeling goals, but there are many other aspects of ML that we haven't covered, and honestly, you just can't cover everything. ML is always evolving, progressing, and branching out, and covers every data domain, which is what makes it so fun! Here we'll briefly discuss some of the other aspects of ML that you'll want to be aware of as you continue your journey.

---

### 9.1 Key Ideas

Some things to keep in mind when thinking about ML as we wrap up our discussion:

- There is practically no modeling or data domain where ML cannot potentially be applied.
- Other widely used techniques include unsupervised settings, reinforcement learning, computer vision, natural language processing, and more generally, artificial intelligence.
- Tabular data has historically been the most common data setting for modeling by far, but this may not always be the case moving forward.

#### 9.1.1 Why this matters

It's very important to know just how unlimited the modeling universe is, but also how there is a tie that binds. Even when we get into other data situations and complex models, we can always fall back on the core approaches we've already seen and know well at this point, and know that those ideas can potentially be applied in any modeling situation.

#### 9.1.2 Good to know

For the stuff in this chapter, a basic idea of modeling and machine learning would be enough.

## 9.2 Unsupervised Learning

All the models considered thus far would fall under the name of **supervised learning**. That is, we have a target variable that we are trying to predict, and we use the data to train a model to predict the target. However, there are settings in which we do not have a target variable, or we do not have a target variable for all of the data. In these cases, we can still use what's often referred to as **unsupervised learning** to learn about the data. Unsupervised learning is a type of machine learning that involves training a model without an explicit target variable in the sense that we've seen. But to be clear, a model is still definitely there! Unsupervised learning attempts learn patterns in the data in a general sense, and can be used in a wide range of applications, including clustering, anomaly detection, and dimensionality reduction, though it's best to think of these as different flavors of a more general approach.

Traditionally, one of the more common applications of unsupervised learning falls under the heading of **dimension reduction**, or **data compression**, such that we reduce features to a smaller **latent**, or hidden, or unobserved, subset that accounts for most of the (co-)variance of the larger set. Alternatively, we reduce the rows to a small number of hidden, or unobserved, clusters. For example, we start with 100 features and reduce them to 10 features that still account for most of what's important in the original set, or we classify each observation as belong to 2-3 clusters. Either way, the primary goal is to reduce the dimensionality of the data, not predict an explicit target.

Classical methods in this domain include principal components analysis (PCA), singular value decomposition (SVD), and factor analysis, which are geared toward reducing column dimensions, as well as cluster methods such as k-means and hierarchical clustering for reducing observations into clusters. Sometimes these methods are often used as preprocessing steps for supervised learning problems, or as a part of exploratory data analysis, but often they are end in themselves. Most of us our familiar with **recommender systems**, whether via Netflix or Amazon, which suggest products or movies, and we're all now becoming extremely familiar with text analysis methods via chat bots. While the underlying models are notably more complex these days, they actually just started off as SVD (recommender systems) or a form of factor analysis (text analysis via latent semantic analysis/latent dirichlet allocation). Having a conceptual understanding of the simpler methods can aid in understanding the more complex ones.

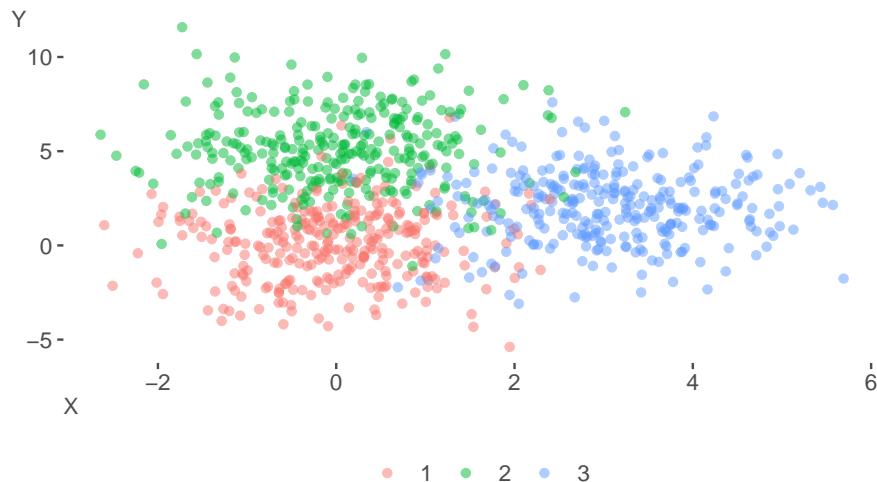


Figure 9.1: Two Variables with Three Overlapping Clusters

💡 In general, do not use a dimension reduction technique as a preprocessing step for a supervised learning problem. Instead, use a supervised learning technique that can handle high-dimensional data, has a built-in way to reduce features (e.g. lasso, boosting), or use a dimension reduction technique that is specifically designed for supervised learning (e.g. partial least squares). Creating a reduced set of features without regard to the target will generally be suboptimal for the supervised learning problem.

### 9.2.1 Connections

#### 9.2.1.1 Clusters are categorical latent features

It turns out that whether we are clustering rows or reducing columns we're actually just using different methods to reduce the features. For methods like PCA and factor analysis, we're reducing the columns to a smaller set of numeric features. For example, we might take answers to dozens of questions of a personality inventory, and reduce them to five key features that represent general aspects of personality. These new features are on their own scale, often standardized, but still reflect the variability originally seen in the original

items to some extent<sup>1</sup>. However, think about a case where we just reduce the features to a single variable, and that variable was categorical. Now you have cluster analysis! You can discretize anything, e.g. from a nicely continuous feature to a coarse couple of categories, and this goes for latent variables as well as those we actually see in our data. For example, if we do a factor analysis with one latent feature, we could either convert it to a probability of some class with an appropriate transformation, or just say that scores higher than some cutoff are in cluster A and the others are in cluster B. Indeed, there is a whole class of clustering models called **mixture models** that do just that, i.e. estimate the latent probability of class membership. The point is that the underlying approach can be conceptually similar, and the bigger difference is how we interpret the results.

### 9.2.1.2 PCA as a neural network

Consider the following neural network, called an **autoencoder**. The goal of an autoencoder is to learn a representation of the data that is smaller than the original data, but can be used to reconstruct the original data. It's trained by minimizing the error between the original data and the reconstructed data. The autoencoder is a special case of a neural network used as a component of many larger architectures, but can be used for dimension reduction in and of itself.

Consider the following setup for such a situation:

- Single hidden layer
- Number of hidden nodes = number of inputs
- Linear activation function

An autoencoder in this case would be equivalent to PCA. In this approach, PCA perfectly reconstructs the original data when considering all components, and so the error would be zero. But that doesn't give us any dimension reduction, so we often only retain a small number of components that capture the data variance by some arbitrary amount.

Neural networks however are not bound to linear activation functions, the size of the inputs or even a single layer, and so they provide a much more flexible approach that can compress the data at a certain layer, but still have very good reconstruction error. Typical autoencoders, would have multiple layers with notably more nodes than inputs. It's not as easily interpretable as typical factor analytic techniques, and we still have to sort out the architecture. However, it's a good example of how the same underlying approach can be used for different purposes.

---

<sup>1</sup>Ideally we'd capture all the variability, but that's not going to happen, and some techniques or results may only capture a relatively small percentage. In our personality example, this could be because the questions don't adequately capture the underlying personality constructs (i.e. an issue of the reliability of instrument), or because personality is just not that simple and we'd need more dimensions.

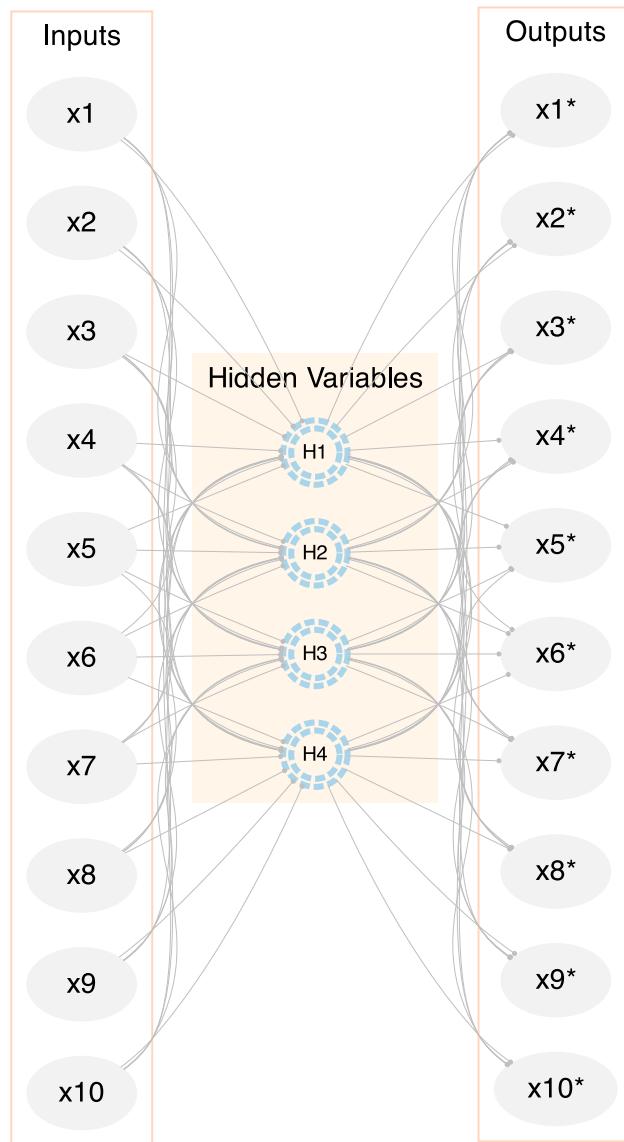


Figure 9.2: PCA or Autoencoder

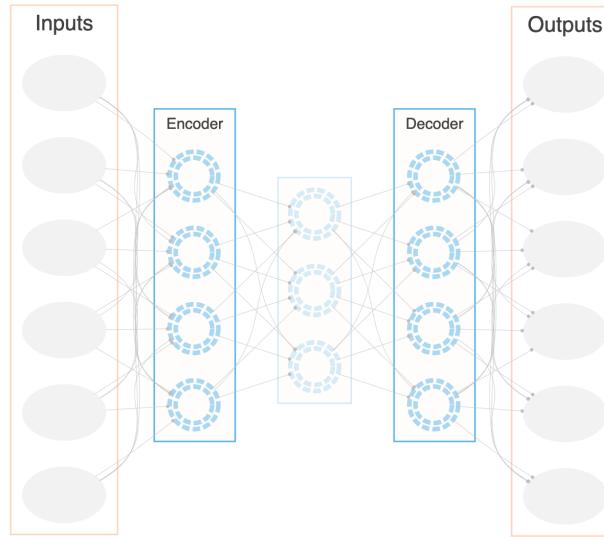


Figure 9.3: Conceptual Diagram of an Autoencoder

TODO: find a way to make graphviz allow for labels/backgrounds on subgraphs with same rank

**i** Autoencoders are special cases of encoder-decoder models, which are used in many applications, including machine translation, image captioning, and more. Autoencoders have the same inputs and outputs, but in other scenarios, a similar type of architecture might be used to classify or generate text, as with large language models.

#### 9.2.1.3 Latent Linear Models

Another thing to be aware of is that factor analytic techniques can be thought of *latent linear models*. Here is a factor analysis as a latent linear model. The ‘targets’ are the observed features, and we predict each one by some linear combination of latent variables.

$$\begin{aligned}x_1 &= \beta_{11}h_1 + \beta_{12}h_2 + \beta_{13}h_3 + \beta_{14}h_4 + \epsilon_1 \\x_2 &= \beta_{21}h_1 + \beta_{22}h_2 + \beta_{23}h_3 + \beta_{24}h_4 + \epsilon_2 \\x_3 &= \beta_{31}h_1 + \beta_{32}h_2 + \beta_{33}h_3 + \beta_{34}h_4 + \epsilon_3\end{aligned}$$

In this scenario, the  $h$  are estimated latent variables, and  $\beta$  are the coefficients, which in some contexts are called **loadings**. The  $\epsilon$  are the residuals, which

are assumed to be independent and normally distributed as with a standard linear model. The  $\beta$  are usually estimated by maximum likelihood, and the model is fit by iterative methods. The latent variables are not observed, but are to be estimated as part of the modeling process, or are derived in post-processing depending on the estimation approach, and typically restricted to have a mean of zero, and possibly standard deviation of 1. The number of latent variables we use is a hyperparameter, and so can be determined by the usual means<sup>2</sup>. To tie some more common models together:

- PCA is a factor analysis with no (residual) variance, and the latent variables are orthogonal (independent).
- Probabilistic PCA is a factor analysis with constant residual variance.
- Factor analysis is a factor analysis with varying residual variance.
- Independent component analysis is a factor analysis that does not assume an underlying gaussian data generating process.
- Non-negative matrix factorization and latent dirichlet allocation are factor analyses applied to counts (think poisson and multinomial regression).

### 9.2.2 Other classical unsupervised learning techniques

There are several techniques that are used to visualize high-dimensional data in a low-dimensional spaces, hopefully to identify clusters or aid with interpretability. These include methods like multidimensional scaling, t-SNE, and (H)DBSCAN. These are often used as a part of exploratory data analysis.

**Cluster analysis** generally speaking has a very long history and you'll see many different approaches, including hierarchical clustering algorithms (agglomerative, divisive), k-means, and more. Distance matrices are often the first step for these clustering approaches, and there are many ways to calculate distances between observations. Conversely, adjacency matrices, which focus on similarity of observations rather than differences, are often used for graph-based approaches, which may also be used for clustering.

**Anomaly/outlier detection** is an approach to find data points of interest. This is often done by looking for data points that are far from the rest of the data, or that are not well explained by the model. This is often used for fraud detection, network intrusion detection, and more. Standard clustering or modeling techniques might be used to identify outliers, or specialized techniques might be used.

**Network analysis** is a type of unsupervised learning that involves analyzing the relationships between entities. It is a graph-based approach that involves

---

<sup>2</sup>Actually, for typical uses of ‘factor analysis’ in a measurement context (including structural equation modeling), e.g. as typically seen in social sciences, cross-validation is pretty very rarely employed, and the number of latent variables is determined by some combination of theory, model comparison for training data only, or trial and error. As a result, one can imagine how reproducible the results are.

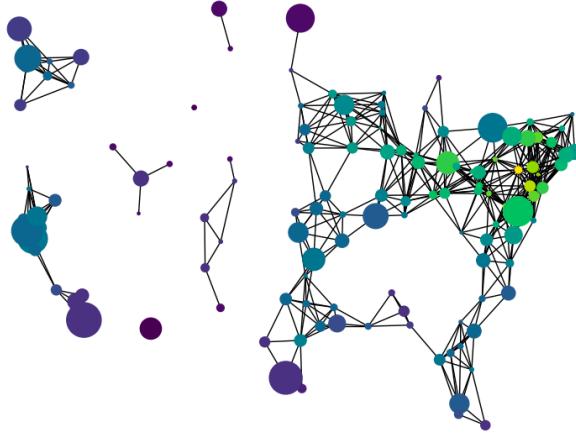


Figure 9.4: Network Graph

identifying nodes (e.g. people) and edges (e.g. do they know each other) in a network. It is used in a wide range of applications, like identifying communities within a network, or to see how they evolve over time. It is also used to identify relationships between entities, such as people, products, or documents. One might be interested in such things as which nodes that have the most connections, or the general ‘connectedness’ of a network. Network analysis or similar graphical models typically have their own clustering techniques that are based on the edge (connection) weights between individuals, such as modularity, or the number of edges between individuals, such as k-clique.

In short, there’s a lot out there that might fall under the umbrella of unsupervised learning, but even when you don’t think you have a target variable, you can still understand or frame these as models similar or even identically to how we have been. One should be less hung up on trying to distinguish modeling approaches with somewhat arbitrary labels, and focus more on what their modeling goal is and how best to achieve it!

### 9.3 Reinforcement Learning

PLACEHOLDER IMAGE

**Reinforcement learning** (RL) is a type of modeling approach that involves training an ‘agent’ to make decisions in an environment. The agent learns by receiving feedback in the form of rewards or punishments for its actions.

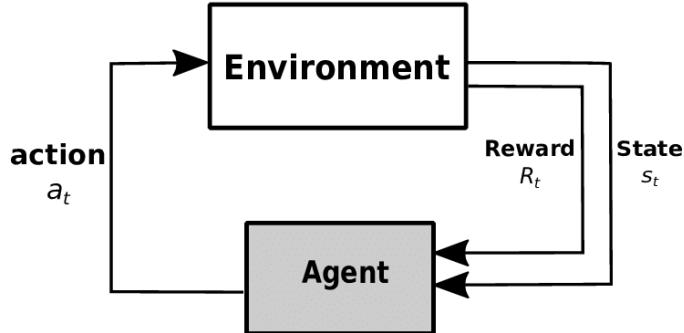


Figure 9.5: Reinforcement Learning

The goal of the agent is to maximize its rewards over time by learning which actions lead to positive or negative outcomes.

In reinforcement learning, the agent interacts with the environment by taking actions and receiving feedback in the form of rewards or punishments. The agent's goal is to learn a **policy**, which is a set of rules that dictate which actions to take in different situations. The agent learns by trial and error, adjusting its policy based on the feedback it receives from the environment. The classic example is a game like chess or simple video games- the agent learns which actions lead to positive outcomes (e.g. winning the game, higher scores) and which actions lead to negative outcomes (e.g. losing the game). The agent then adjusts its policy based on the feedback it receives from the environment. A key aspect of RL is the balance between **exploration and exploitation**, i.e. trying new things that might lead to greater rewards vs. sticking with what works.

Reinforcement learning has many applications, including robotics, game playing, and autonomous driving, but there is little restriction on where it might be applied. It is often a key part of some deep learning models, where reinforcement is supplied via human feedback or other means. In general, RL is a powerful tool that might be useful where traditional programming approaches may not be as feasible.

## 9.4 Non-Tabular Data Applications

While our focus in this book is on tabular data due to its ubiquity, there are many other types of data that can be used for modeling, some of which can still potentially be used in that manner, but which often start as a different

format or must be considered in a special way. Here we'll briefly discuss some of the other types of data you'll potentially come across.

### 9.4.1 Spatial

Spatial data such as geographic information can sometimes be quite complex. Oftentimes it is housed in its own data format (e.g. shapefiles), and there are many specialized tools for working with it. Spatial specific features may include continuous types such as latitude and longitude, or the telemetry of a person's movements recorded from a watch. Others are more discrete such as states within a country. In general, we'd used these features as we would others in the tabular setting, but we often want to take into account the uniqueness of a particular region or the correlation of spatially regions. Historically, most spatial data can be incorporated into models like mixed models or generalized additive models, but in certain applications, such as satellite imagery, deep learning models are more the norm, and the models often transition into image processing techniques.

### 9.4.2 Audio

Audio data is a type of time series data that is also the focus for many modeling applications. It is often represented as a waveform, which is a plot of the amplitude of the sound wave over time. The goal of modeling the data may include speech recognition, music generation, and more. This sort of data, like spatial data, is typically housed in specific formats, and is often of a very large size. Also like spatial data, the specific type and research question may allow for a tabular format, and the modeling approaches are similar to those for other time series data. As in other domains where the data is of a singular type at its core, deep learning has proved very useful, and can even create songs people actually like, even recently helping the Beatles to release one more song<sup>3</sup>.

### 9.4.3 Image Processing

#### DL CONVNETS IMAGE PLACEHOLDER

Image processing involves a range of models and techniques for analyzing images. These include image classification, object detection, image segmentation, tracking, and more. Image classification is the task of assigning a label to an image. Object detection involves identifying the location of objects in an image. Image segmentation is the task of identifying the boundaries of objects in an image. Tracking requires following objects over time.

In general, your base data is an image, which is represented as a matrix of

<sup>3</sup>[https://en.wikipedia.org/wiki/Now\\_and\\_Then\\_\(Beatles\\_song\)](https://en.wikipedia.org/wiki/Now_and_Then_(Beatles_song))

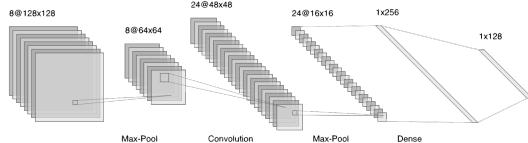


Figure 9.6: Convolutional Neural Network

pixel values. For example, each row of the matrix could be a grayscale value for a pixel, or it could be a vector of RGB values for a pixel, such that each row is an image, the matrix is a collection of images, while the third dimension is the color channel of red, green and blue. The modeling goal then is to extract features from the image that can be used for the task at hand. For example, you might extract features such as color, texture, and shape. You can then use these features to train a model to classify images or whatever your task may be.

Image processing is a broad field with many applications. It is used in medical imaging, satellite imagery, self-driving cars, and more. And while it can be really fun to classify objects such as cats and dogs, or generate images from text and vice versa, it can be quite challenging due to the size of the data, issues specific to video/image quality, and the model complexity. Even if your base data is often the same or very similar, the model architecture and training process can vary widely depending on the task at hand.

#### 9.4.4 Natural Language Processing

##### SOME SORT OF CHAT RELATED IMAGE

**Input**

[Text from: <https://en.wikipedia.org/wiki/Rihanna>]

Describe her Super Bowl performance and what made it special.

---

**Output**

Rihanna's Super Bowl LVII halftime show performance was special for a number of reasons. For one, it marked her first live performance in over five years, and it was her first time headlining the Super Bowl halftime show. Additionally, she made history by becoming the first person to perform at the Super Bowl while pregnant. The show itself was critically acclaimed, with many praising her high-energy performance and impressive vocals. It also garnered a massive viewership, becoming the second most-watched halftime show in history. ■

Figure 9.7: Demo for GPT4

It's safe to say that the hottest area of modeling development in recent times regards **natural language processing**, as evidenced by the runaway success of models like ChatGPT. Natural language processing (NLP) is a field of study that focuses on understanding human language, and can be seen as a very visible subfield of artificial intelligence. NLP is used in a wide range of applications, including machine translation, speech recognition, text classification, and more. NLP is behind some of the most exciting applications today, with tools that continue to amaze with their capabilities to generate summaries of articles, answering questions, write code, and even pass the bar exam with flying colors<sup>4</sup>!

Early efforts in this field were based on statistical models, and then variations on things like PCA, but it took a lot of data pre-processing work<sup>5</sup> to get much from those approaches, and results could still be unsatisfactory. However, more recently, deep learning models have become the standard application, and there is no looking back in that regard. Current state of the art models have been trained on massive amounts of data, even the entire internet, and can be used for a wide range of tasks. But you don't have to train such a model yourself- now you can simply use a pre-trained model like GPT-4 for many NLP tasks, and in some cases much of the trouble comes with generating the best prompt to produce the desired results. However, the field and the models are evolving extremely rapidly, and things are getting easier all the time<sup>6</sup>.

#### 9.4.5 Pre-trained Models & Transfer Learning

Pre-trained models are models that have been trained on a large amount of data, and can be used for a wide range of tasks. They are widely employed in image and natural language processing. The basic idea is that, if you can use a model that was trained on the entire internet of text, why start from scratch? Image processing models already understand things like edges and colors, so there is little need to reinvent the wheel when you know those features would be useful for your own task. These are viable in tasks where the inputs are similar to the data the model was trained on, as is the case with images and text.

You can use a pre-trained model as a starting point for your own model, and then **fine-tune** it for your specific task, and this is more generally called **transfer learning**. The gist is that you only need to train part of the model on your specific data, or possibly even not at all. You can just feed your data in and get predictions from the ready-to-go model! This obviously can save a lot of time and resources, assuming you don't have to pay much to use the

<sup>4</sup><https://www.abajournal.com/web/article/latest-version-of-chatgpt-aces-the-bar-exam-with-score-in-90th-percentile>

<sup>5</sup><https://m-clark.github.io/text-analysis-with-R/intro.html>

<sup>6</sup>It seems unlikely the prompt engineering will still be something of interest in a couple years, at least, probably not enough to warrant whole courses for it.

model in the first place, and can be especially useful when you don't have a lot of data to train your model on.

<https://bbycroft.net/llm>

#### 9.4.6 Combining Models

It's also important to note that these types of data and their associated models are not mutually exclusive. For example, you might have a video that contains both audio and visual information pertinent to the task. Or you might want to produce images from text inputs. In these cases, you can use a combination of models to extract features from the data, which may just be more features in a tabular format, or be as complex as a multimodal deep learning architecture. Many vision, audio, natural language and other modeling approaches incorporate **transformers**. They are based on the idea of **attention**, which is a mechanism that allows the model to focus on certain parts of the input sequence and less on others. Transformers are used in many state-of-the-art models with different data types such as those that combine text and images. The transformer architecture is a bit complex, but it's worth knowing about as it's used in many of the most advanced models today.

---

## 9.5 Artificial Intelligence

The prospect of combining models for computer vision, natural language processing, audio processing, and other domains can produce tools that mimic many aspects of what we call intelligence<sup>7</sup>. Current efforts in AI produce models that can pass law and medical exams, create better explanations of images and text than average human effort, and produce conversation on par with humans. AI even helped to create this book!

In many discussions of ML and AI, many put ML as a subset of AI<sup>8</sup>, but this is a bit off the mark from a modeling perspective in our opinion<sup>9</sup>. For example, model-wise, any aspect of what we'd call modern AI almost exclusively employs deep learning models (although it didn't in the past, and may supplement a DL model with non-DL models), while the ML approach to training and evaluating models can be used for any underlying model, from

<sup>7</sup>It seems most discussions of AI in the public sphere never bother to define intelligence very clearly in the first place, and the academic realm has struggled with the concept for centuries.

<sup>8</sup><https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/artificial-intelligence-vs-machine-learning>

<sup>9</sup>Almost every instance of this we've seen also never goes into actual detail or specific enough definitions to make the comparison meaningful to begin with, so don't take it too far.



Figure 9.8: AI

simple linear models to the most complex deep learning models, whether the application falls under the heading of AI or not. Furthermore, statistical model applications have never seriously attempted what we might call AI. If AI is some ‘autonomous and general tools that attempt to engage the world in a human-like way or better’, it’s not clear why it’d be compared to ML in the first place. That’s kind of like saying the brain is a subset of cognition. The brain does the work, much like ML does the modeling work with data, and gives rise to what we call cognition, but generally we would not compare the brain to cognition. The point is that to not get too hung up on the labels, and focus on the modeling goal and how best to achieve it. Deep learning models, and machine learning in general, can be used for non-AI settings, as we have seen for ourselves. And models still employ the *perspective* of the ML approach when ultimately used for AI - the steps taken from data to model output are largely the same.

Many of the non-AI settings we use modeling for may well be things we can eventually rely on AI to do, but the computational limits, the amount of data that would be required for AI models to do well, or the ability of AI to be able to deal with situations in which there is *only* small bits of data, are still hinderances in current applications of AI. However, we feel it’s likely these will eventually be overcome. But even then, a statistical approach may still have a place when the data does become small, e.g., with model comparison.

**Artificial general intelligence (AGI)** is the holy grail of AI, and like AI

itself is not consistently defined. In general, the idea behind AGI is the creation of some autonomous agent that can perform any task that a human can perform, many that humans cannot, and generalize abilities to new problems that have not even been seen yet. It seems we are getting closer to AGI all the time, but it's not yet clear when it will be achieved, or even what it will look like when it is achieved, especially since no one has an agreed upon definition of what intelligence is in the first place.

That said, to be frank, you may very likely be reading a history book. Given recent advancements just in the last year or so, it seems unlikely that the data science being performed five years from now will resemble much of how things are done today<sup>10</sup>. We are already capable of making faster and further advancements to do AI, and it's likely that the next generation of data scientists will be able to do so even more easily. The future is here, and it is amazing. Buckle up!

---

## 9.6 Where to go from here

The sky is the limit with machine learning and modeling. Go where your heart leads you, and have some fun! But if you want some more guidance, here are some ideas:

### TODO: NEEDS MORE WORK

- Courses on ML and DL: fastai, coursera, etc.
- Kaggle competitions
- Read papers
- Do more modeling!

#### 9.6.0.1 refs

Rashcka <https://nostarch.com/machine-learning-and-ai-beyond-basics>

Vaswani, Ashish; Shazeer, Noam; Parmar, Niki; Uszkoreit, Jakob; Jones, Llion; Gomez, Aidan N; Kaiser, Łukasz; Polosukhin, Illia (2017). “Attention is All you Need” (PDF). Advances in Neural Information Processing Systems. Curran Associates, Inc. 30.

Unsupervised: [<https://cloud.google.com/discover/what-is-unsupervised-learning>]

Visuals and deeper understanding: <https://colah.github.io/>

---

<sup>10</sup>A good reference for this sentiment is a scene from Star Trek in which Scotty has to use a contemporary computer<sup>11</sup>.

embeddings  
Representations/

[https://colah.github.io/posts/2014-07-NLP-RNNs-](https://colah.github.io/posts/2014-07-NLP-RNNs-Representations/)

---

---

## Part III

# Other Considerations



# 10

---

## *Data Issues in Modeling*

---

It's an inescapable fact that models need data to work, even if it's simulated data. In this chapter we'll discuss some of the most common data issues, with brief overviews so that you have an idea of why you'd care. There's a lot to know about data in general before you ever get into modeling your own, so we'll give you some things to think about it in this chapter.

---

### 10.1 Key Ideas

- Data transformations can provide many modeling benefits.
  - Categorical data still needs a numeric representation, and this can be done in a variety of ways.
  - The data type for the target may suggest a particular model, but does not necessitate one.
  - The data *structure*, e.g. temporal or structural, likewise may suggest a particular model.
  - Latent variables are everywhere!
- 

### 10.2 Standard Feature & Target Transformations

Transforming variables can provide several benefits in modeling, whether applied to the target, covariates, or both, and *should regularly be used for most model situations*. Some of these benefits include:

- Interpretable intercepts
- More comparable covariate effects
- Faster estimation
- Easier convergence
- Help with heteroscedasticity

For example, merely centering predictor variables, i.e. subtracting the mean, provides a more interpretable intercept that will fall within the actual range

of the target variable, telling us what the value of the target variable is when the covariates are at their means (or reference value if categorical). But even if easier interpretation isn't a major concern, variable transformations can help with convergence and speed up estimation, so can always be of benefit.

### 10.2.1 Numeric variables

The following table shows the interpretation of some very common transformations applied to numeric variables- logging and standardization, (i.e. standardizing to mean zero, standard deviation one).

Table 10.1: Common numeric transformations

Target	Feature	Change in X	Change in Y	Benefits
y	x	1 unit	B unit	Interpretation
log(y)	x	1 unit	100 * (exp(B) -1)	Heteroscedasticity in y
log(y)	log(x)	1% change	B% change	Interpretation, deal with feature extremes
y	scale(x)	1 standard deviation	B unit	Interpretation, estimation
scale(y)	scale(x)	1 standard deviation	B standard deviation	Interpretation, estimation

For example, it is very common to use **standardized** variables, or simply ‘scaling’ them. Some also call this **normalizing** but this can mean a lot of things<sup>1</sup>, so one should be clear in their communication. If  $y$  and  $x$  are both standardized, a one unit (i.e. one standard deviation) change in  $x$  leads to a  $\beta$  standard deviation change in  $y$ . Again, if  $\beta$  was .5, a standard deviation change in  $x$  leads to a half standard deviation change in  $y$ . In general, there is nothing to lose by standardizing, so you should employ it often.

Another common transformation, particularly in machine learning, is **min-max scaling**, changing variables to range from some minimum to some maximum, which is almost always zero to one. This can make numeric and categorical indicators more comparable, or at least put them on the same scale for estimation purposes, and so can help with convergence and speed up estimation.

#### 10.2.1.1 Python

When using `sklearn` it's a very verbose process to do a simple transformation, but this is beneficial when you want to do more complicated things, especially when using data pipelines.

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler
import numpy as np

# Create a sample dataset
```

<sup>1</sup>[https://en.wikipedia.org/wiki/Normalization\\_\(statistics\)](https://en.wikipedia.org/wiki/Normalization_(statistics))

```

import numpy as np

# Create a random sample of integers
data = np.random.randint(low=0, high=100, size=(5, 3))

# Apply StandardScaler
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data)

# Apply MinMaxScaler
minmax_scaler = MinMaxScaler()
minmax_scaled_data = minmax_scaler.fit_transform(data)

```

### 10.2.1.2 R

R being made for statistics, it's much easier to do simple transformations, but you can also use tools like `recipes` to and `mlr3` pipeline operations when needed to make sure your preprocessing is applied appropriately.

```

# Create a sample dataset
data = matrix(sample(1:100, 15), nrow = 5)

# Standardization
scaled_data = scale(data)

# Min-Max Scaling
minmax_scaled_data = apply(data, 2, function(x) {
  (x - min(x)) / (max(x) - min(x))
})

```

Using a **log** transformation for numeric targets and features is straightforward, and comes with several benefits<sup>2</sup>. For example, it can help with **heteroscedasticity**, i.e. when the variance of the target is not constant across the range of the predictions<sup>3</sup> (demonstrated below), keeping predictions positive after transformation, allows for interpretability gains, and more. One issue with logging is that it is not a linear transformation, and so can make certain more complicated transformations in post-modeling more less straightforward. Also if you have a lot of zeros, log plus one transformations are not going to be enough to help you overcome that hurdle. It also won't help much when the variables in question have few distinct values, like ordinal variables, which we'll discuss later in Section 10.2.3.

---

<sup>2</sup><https://stats.stackexchange.com/questions/107610/what-is-the-reason-the-log-transformation-is-used-with-right-skewed-distribution>

<sup>3</sup>For the bazillionth time, logging does not make data 'normal' so that you can meet your normality assumption in linear regression. The only time that would work is if your data is log-normally distributed to begin with.

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.5323	1.8196	2.7330	3.1763	4.0295	12.1670

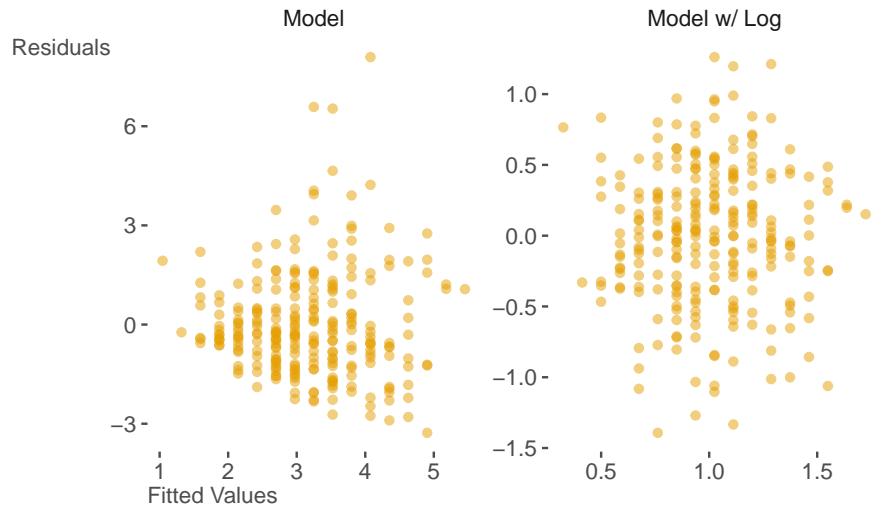


Figure 10.1: Log transformation and heteroscedasticity



It is rarely necessary or a good idea to transform a numeric feature to a categorical one. This is because you are potentially throwing away useful information by making the feature a less reliable measure of the underlying construct. For example, discretizing age to ‘young’ and ‘old’ does not help your model, and you can always get predictions for what you would consider ‘young’ and ‘old’ after the fact. The primary reasons for doing this are not statistically sound, and can actually hinder interpretation by creating arbitrary groups.

### 10.2.2 Categorical variables

A raw character string is not an analyzable unit, so character strings and labeled variables like factors must be converted for analysis to be conducted on them. For categorical variables, we can employ what is called **effects coding** to test for specific types of group differences. Far and away the most common approach is called **dummy coding** or **one-hot encoding**<sup>4</sup>. In these situations we create columns for each category, and the value of the column is 1 if

---

<sup>4</sup>Note that one-hot encoding can refer to just the 1/0 coding for all categories, or to the specific case of dummy coding where one category is dropped. Make sure the context is clear.

the observation is in that category, and 0 otherwise. Here is a one-hot encoded version of the `season` feature.

Table 10.2: One-hot encoding

seasonFall	seasonSpring	seasonSummer	seasonWinter	season
1.00	0.00	0.00	0.00	Fall
1.00	0.00	0.00	0.00	Fall
1.00	0.00	0.00	0.00	Fall
1.00	0.00	0.00	0.00	Fall
0.00	0.00	1.00	0.00	Summer
0.00	0.00	1.00	0.00	Summer

**i** When doing statistical models, when doing one-hot encoding all relevant information is incorporated in  $k-1$  groups, so one category will be dropped. For example, in a linear model, the intercept is the mean of the target for the dropped category, and the coefficients for the other categories are the difference between the mean for the dropped, a.k.a. **reference**, category and the mean the category being considered. As an example, in the case of the `season` feature, if the dropped category is `winter`, the intercept tells us the mean rating for `winter`, and the coefficients for the other categories are the difference between the value for `winter` and the mean of the target for the included category. For other modeling approaches, all categories are included, and the model will learn the best way to use them, and may even only consider some or one of them at a particular iteration of estimation.

Another important way to encode categorical information is through an **embedding**. This is a way of representing the categories as a vector of numbers, at which point the embedding feature is used in the model like anything else. The way to do this usually involves a model itself, one that learns the best way to represent the categories as numbers. This is a very common approach in deep learning, and can be done simultaneously with the rest of the model, but can potentially be used in any modeling situation as an initial data processing step.

### 10.2.3 Ordinal Variables

So far in our discussion, our categorical data has been assumed to have no order. However you may find yourself with orders labels like “low”, “medium”, and “high”, or “bad” ... to “good”, or simply are a few numbers, like ratings from 1 to 5. **Ordinal data** is categorical data that has a known ordering, but

which still has arbitrary labels. Let us repeat that, *ordinal data is categorical data.*

#### 10.2.3.1 Ordinal Features

The simplest way to treat ordinal features is as if they were numeric. If you do this, then you're just pretending that it's not categorical, and this is usually fine. Most of the transformations we mentioned probably aren't going to be as useful, but you can still use them if you want. For example, logging five values of ratings 1-5 isn't going to do anything for you, but it technically doesn't hurt anything. But you should know that typical statistics like means and standard deviations don't really make sense for ordinal data, so the main reason for treating them as numeric is for modeling convenience.

If you choose to treat it as categorical, you can ignore the ordering and do the same as you would with categorical data. There are some specific approaches to coding ordinal data for use in linear models, but they are not common, and they generally aren't going to help the model or interpreting it, so we do not recommend them. You could however use old-school **contrast encodings** that you would in traditional ANOVA approaches, but again, you'd need a good reason to do so.

Take home message: treat ordinal features as you would numeric or non-ordered categorical. Either is fine.

#### 10.2.3.2 Ordinal Targets

Ordinal targets are a bit more complicated. If you treat them as numeric, you're assuming that the difference between 1 and 2 is the same as the difference between 2 and 3, and so on. This is probably not true. If you treat them as categorical, you're assuming that there is no connection between categories, e.g. that in order to get to category three you have to have gone through category 2. So what should you do?

There are a number of approaches to modeling ordinal targets, but the most common is the **proportional odds model**. This model can be seen as a generalization of the logistic regression model, and is very similar to it, and actually identical if you only had two categories. But others are also possible, and your results could return something that gives coefficients for the model for the 1-2 category change, the 2-3 category change, and so on.

Ordinality of a categorical outcome is largely ignored in machine learning approaches. The outcome is either treated as numeric or multi-category classification. This is not necessarily a bad thing, especially if prediction is the primary goal.

💡 Some are a little too eager to jump to simultaneously modeling multiple target variables, e.g. in structural equation modeling or multivariate regression. It's not wrong to do so, but given the difficulty our brains have with interpreting results for a single target, you might think twice about doing so. However, for scenarios focused much more on prediction performance that involve methods like deep learning, it makes more sense, and is actually required.

---

### 10.3 Missing Data

Missing data is a common challenge in data science, and there are a number of ways to deal with it, usually by substituting, or **imputing** some value for the missing one. The most common approaches are:

- **Complete case analysis:** Only use observations that have no missing data.
- **Single value imputation:** Replace missing values with the mean, median, mode or some other value of the feature.
- **Model-based imputation:** Use a model based on complete cases to predict the missing values, and use those predictions as the imputed values.
- **Multiple imputation:** Create multiple imputed datasets based on the predictive distribution of the model used in model-based imputation. Estimates of coefficients and variances are averaged in some fashion over the imputations.
- **Bayesian imputation:** Treat the missing values as parameters to be estimated.

The first approach to drop missing data is the simplest, but can lead to a lot of lost data, and can lead to biased statistical results if the data is not **missing completely at random**. There are special cases of some models that by their nature can ignore the missingness under an assumption of **missing at random**, but even those models would likely benefit from some sort of imputation. If you don't have much missing data, this would be fine. How much is too much? Unfortunately that depends on the context, but if you have more than 10% missing, you should probably be looking at alternatives.

The second approach where we just plug in a single value like a mean is also simple, but will probably rarely help your model. Consider a numeric feature that is 50% missing, and for which you replace the missing with the mean. How good do you think that feature will be when at least half the values are identical? Whatever variance it normally would have and share with the target is probably reduced, and possibly dramatically. Furthermore, you've also attenuated correlations it has with the other features, which could

potentially further hamper interpretation or cause other issues depending on the type of model you're implementing. Single value imputation makes perfect sense if you know that the missingness means a specific value, like a count feature where missing means a count of zero. If you don't have much missing data, it's unlikely this would have any real benefit over complete case analysis, *except* if it allows you to use all the other features that would otherwise be dropped. But then, why not just drop this feature and keep the others?

Model-based imputation is more complicated, but can be very effective. In essence, you run a model for complete cases in which the target is now the feature with missing values, and the covariates are all the other features and target. You then use that model to predict the missing values, and use those predictions as the imputed values. After these predictions are made, you move on to the next feature and do the same. There are no restrictions on which model you use for which feature. If the other features in the imputation model also have missing data, you can use something like mean imputation to get more complete data if necessary as a first step, and then when their turn comes, impute those values.

Although the implication is that you would have one model per feature and then be done, you can do this iteratively for several rounds, such that the initial imputed values are then used in subsequent models to reimpute the missing values. You can do this as many times as you want, but the returns will diminish.

Multiple imputation (MI) is the most complicated, but can be the most effective under some situations, depending on what you're willing to sacrifice for having better uncertainty estimates vs. a deeper dive into the model. The idea is that you create multiple imputed datasets, each of which is based on the **predictive distribution** of the model used in model-based imputation. Say we use a linear regression assuming a normal distribution to impute feature A. We would then draw from the predictive distribution of that model to create a dataset with imputed values for feature A, then do it again, say a total of 10 times.

You now have 10 imputed data sets. You then run your actual model of interest on each of these datasets, and your final model results are a kind of average of the parameters of interest (or exactly an average, say for regression coefficients). This main thing this approach provides is that it acknowledges that your single imputation methods have uncertainty in those model predictions being used as imputed values, and that uncertainty is incorporated into the final model results.

MI can in theory handle the any source of missingness and as such is a very powerful approach. But it has several drawbacks. One is that you need statistical or generative model and distribution for all models used, and that distribution is something you have to assume is appropriate. Your final model

presumably is also a probabilistic model with coefficients and variances you are trying to estimate and understand. MI isn't really going to help an XGBoost or deep learning model for example, or at least offer little if anything over single value imputation. If you have very large data and a complicated model, you could be waiting a long time, and as modeling is an iterative process itself, this can be rather tedious to work through. Finally, few data or post-model processing tools that you commonly use will work with MI results, especially visualization ones, and so you will have to hope that whatever package you use for MI has what you need. As an example, you'd have to figure out how you're going to impute interaction terms if you have them, practically nothing will work with cross-validation approaches,

*Practically speaking*, MI takes a lot of effort to often come to the same conclusions you would have with a single imputation approach, or possibly fewer conclusions for anything beyond GLM coefficients and their standard errors. But if you want your uncertainty estimate for those models to be better, MI can be an option.

One final option is to run a Bayesian model where the missing values are treated as parameters to be estimated. MI basically is a poor man's Bayesian imputation approach. A package like `brms` can do this, and it can be very effective, but it is also very computationally intensive, and can be very slow. At least it would be more fun than standard MI!

---

## 10.4 Class Imbalance

**Class imbalance** refers to the situation where the target variable has a large difference in the number of observations in each class. For example, if you have a binary target, and 90% of the observations are in one class, and 10% in the other, you would have class imbalance. You'll almost never see a 50/50 split in the real world, but the issue is that as we move further away from that point, we can start to see problems in model estimation, prediction, and interpretation. As a starting point, if we just predict the majority class in a binary classification problem, we'll be right 90% of the time in terms of accuracy. So right off the bat one of our favorite metrics isn't going to help us assess model performance as much as we'd like.

**In classification problems, class imbalance is the rule, not the exception.** This is because nature just doesn't sort itself into nice and even bins. For example, the majority of people of a random sample do not have cancer, the vast majority of people have not had a heart attack in the past year, most people do not default on their loans, and so on.

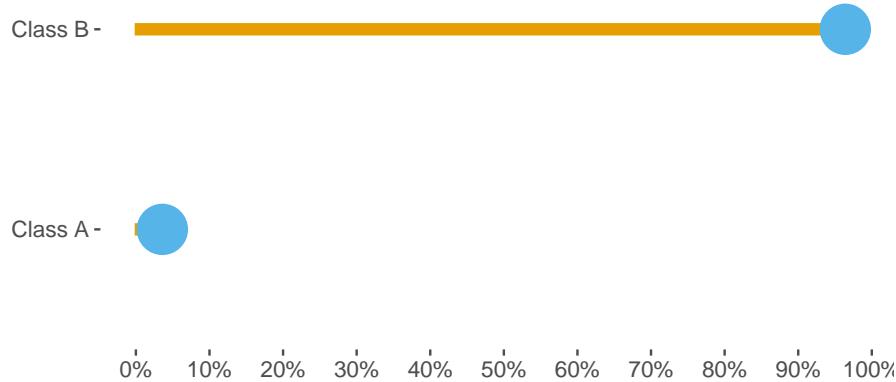


Figure 10.2: Class Imbalance

There are a number of ways to help deal with class imbalance, and the best approach depends on the context. Some of the most common approaches are:

- **Use different metrics:** Use metrics that are less affected by class imbalance, such as area under a receiver operating characteristic curve (AUC), or those that balance the
- **Oversampling/Undersampling:** Randomly sample from the minority (majority) class to increase (decrease) the number of observations in that class.
- **Weighted objectives:** Weight the loss function to give more weight to the minority class. Although commonly employed, and a simple thing to use with models like lightgbm and xgboost, it often fails to help, and can cause other issues.
- **Thresholding:** Change the threshold for classification to be more sensitive to the minority class. Nothing says you have to use 0.5 as the threshold for classification, and you can change it to be more sensitive to the minority class. This is a very simple approach, and may be all you need.

These are not necessarily mutually exclusive. For example, it's probably a good idea to switch to a metric besides accuracy as you employ other techniques.

### 10.4.1 Calibration issues in classification

Probability **calibration** is often an issue in classification problems, and is a bit more complicated than just class imbalance but is often discussed in the same setting. Having calibrated probabilities refers to the situation where the predicted probabilities of the target match up well to the actual probabilities. For example, if you predict that 10% of people will default on their loans, and 10% of people actually do default on their loans, one would say your model is well calibrated. Conversely, if you predict that 10% of people will default on their loans, but 20% of people actually do default on their loans, your model is not so well-calibrated.

One of the most common approaches to assessing calibration is to use a **calibration curve**, which is a plot of the predicted probabilities vs. the observed proportions. In the following, one model seems to align well with the observed proportions based on the chosen bins. The other model is not so well calibrated, and is overshooting with its predictions.

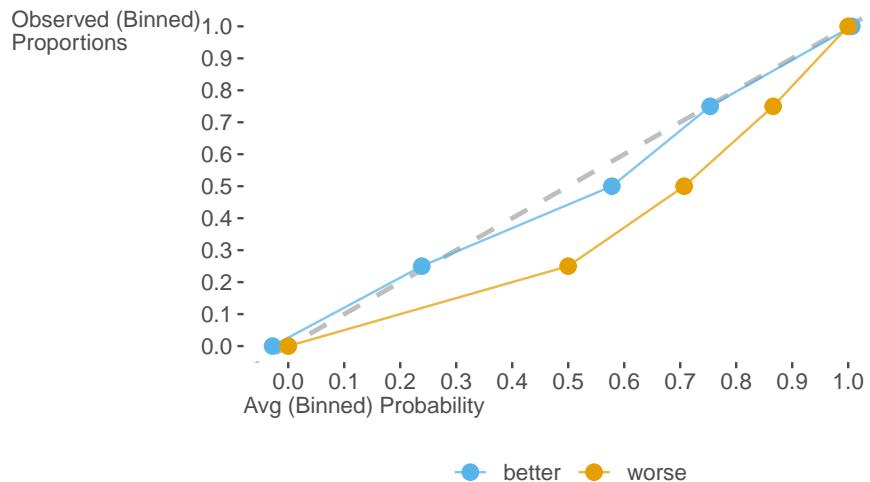


Figure 10.3: Calibration Plot

While the issue is an important one, it's good to keep the issue of calibration and imbalance separate. As miscalibration implies bias, bias can happen irrespective of the class proportions and can be due to a variety of factors related to the model, target, or features, and miscalibration is not inherent to a particular model.

Furthermore, the assessment of calibration with this approach has a few issues. For one, the observed ‘probabilities’ are proportions based on arbitrarily

chosen bins and observed values that are measured with some measurement error as well as having natural variance will partly reflect sample size<sup>5</sup>. These plots are often presented such that observed proportions are labeled as the “true” probabilities. However, you do not have the *true* probabilities outside of simulation settings, just the observed class labels, so whether your model’s predicted probabilities match observed proportions is a bit of a different question. The predictions obviously have uncertainty as well, and this will depend on modeling approach, sample size, etc. And finally, the number of bins chosen can also affect the appearance of the plot in a notable way.

All this is to say that each point in a calibration plot has some error bar around it, and the differences between models and the ‘best case scenario’ would need additional steps to suss out. Some methods are available to calibrate probabilities, but they are not commonly implemented in practice, and often involve a model-based technique, with all of its own assumptions and limitations. It’s also not exactly clear that forcing your probabilities to be on the line is helping solve the actual modeling goal in any way<sup>6</sup>. But if you are interested, you can read more here<sup>7</sup>.

---

## 10.5 Censoring and Truncation

Sometimes, we just don’t see all the data there is to see. **Censoring** is a situation where the target variable is not fully observed. This is common in survival analysis<sup>8</sup>, where the target is the time to an event, but the event has not yet occurred for some observations. This is called **right censoring**, and is the most common type of censoring, and depicted in the above plot, where several individuals are only observed to a certain age and were still alive at that time. There is also **left censoring**<sup>9</sup>, where the censoring happens from the other direction data before a certain point is unknown. Finally, there is **interval censoring**, where the event of interest occurs within some interval, but the exact value is unknown.

Survival analysis is a common modeling technique in this situation, but you may also be able to keep things even more simple via something like tobit

---

<sup>5</sup>Recall that each bin will only be portion of the test set size.

<sup>6</sup>Oftentimes we are only interested in the ordering of the predictions, and not the actual probabilities. For example, if we are trying to identify the top 10% of people most likely to default on their loans, we’ll just take the top 10% of predictions, and the actual probabilities are irrelevant for that goal.

<sup>7</sup><https://scikit-learn.org/stable/modules/calibration.html>

<sup>8</sup>Survival analysis is also called event history analysis, and is a very common approach in biostatistics, sociology, demography, and other disciplines where the target is the time to an event, such as death, marriage, divorce, etc.

<sup>9</sup><https://stats.stackexchange.com/questions/144037/right-censoring-and-left-censoring>

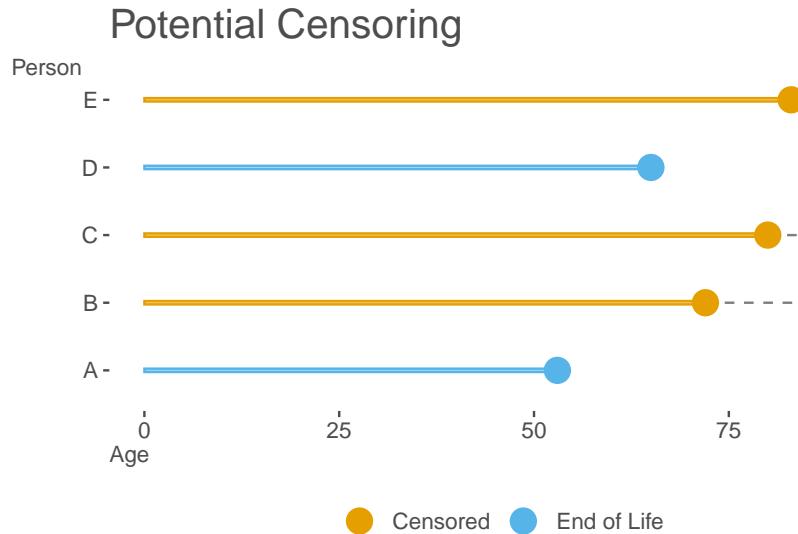


Figure 10.4: Potential Censoring

regression<sup>10</sup>. In this approach, you assume that the target is fully observed, but that the values are censored, and you model the probability of censoring. This is a very common approach in econometrics, and can keep you in a traditional linear model context.

**Truncation** is a situation where the target variable is only observed if it is above or below some value. One of the issues is that default distributional methods, e.g., via maximum likelihood, assume a distribution that is not necessarily bounded. In our plot above, we restrict our data to 70 and below, but typical modeling methods with default distributions would not respect that.

You could truncate predictions after the fact, but this is a bit of a hack, and often results in lumpiness in the predictions at the boundary that isn't realistic in most situations. Alternatively, Bayesian approaches allow you to model the target as a distribution with truncated distributions, and so you can model the probability of the target being above or below some value. This is a very flexible approach. There are also models such as hurdle models that might prove useful where the truncation is theoretically motivated, e.g. a zero-inflated Poisson model for count data where the zero counts are due to a separate process than the non-zero counts.

---

<sup>10</sup><https://m-clark.github.io/models-by-example/tobit.html>

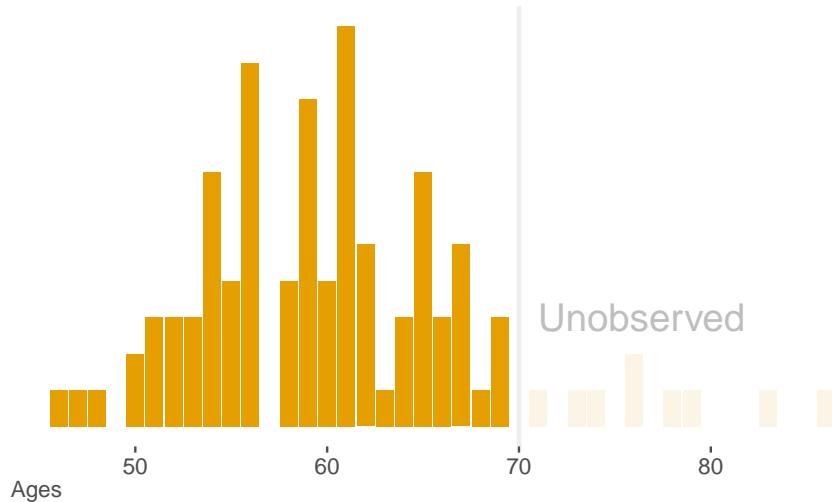


Figure 10.5: Truncation

**i** One way to distinguish censored and truncated data is that censored data is usually due to some external process such that the target is not observed but could be possible (capping reported income at \$1 million), whereas truncated data is due to some internal process that prevents the target from being observed, and is often derived from sample selection (we only want to model non-millionaires). We would not want observations past the censored point to be unlikely, but we would want observations past the truncated point to be impossible. Trickier still is that for bounded or truncated distributions that might be applied to the truncated scenario, such as folded vs. truncated distributions, they would not result in the same probability distributions even if they can be applied to the same situation.

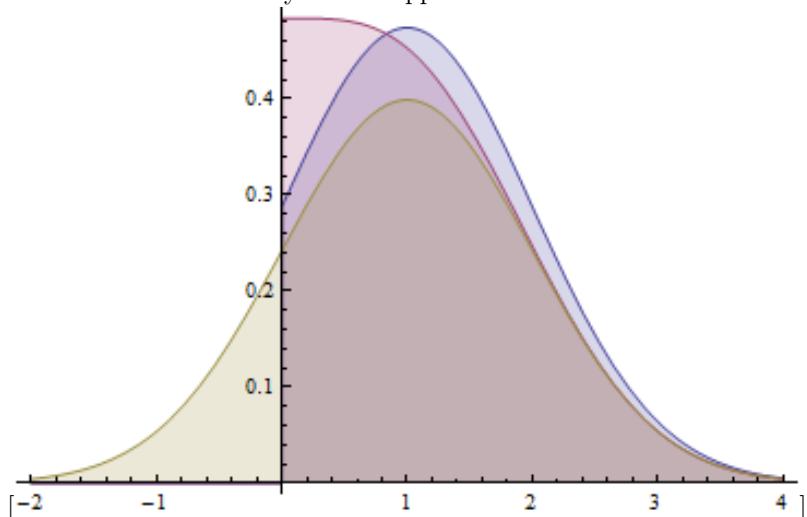


Image from StackExchange<sup>a</sup>

<sup>a</sup>(<https://stats.stackexchange.com/questions/16089/is-sampling-from-a-folded-normal-distribution-equivalent-to-sampling-from-a-norm>)

## 10.6 Time Series

TODO: ADD CHAPTER LINKS HERE

Time series data is any data that incorporates values over time. This could be something like the a state's population over years, or the max temperature of an area over days, etc. Time series data is very common, and there are a number of approaches to modeling it, and we usually take special approaches

to account for the fact that observations are not independent of one another. The most common approach is to use a **time series regression** approach, where the target is some value that varies over time, and the covariates are other features that can be time-varying or not. There are old school approaches like **ARIMA** that can still serve as decent [baseline models]LINK THE TO ML CHAPTER, and newer approaches like more sophisticated Bayesian ones that can get quite complex.

**Longitudinal data**<sup>11</sup> is a special case of time series data, where the target is a function of time, but it is typically grouped in some fashion. An example would be school performance for students over several semesters, where values are clustered within students over time. In this case, you can use a time series regression approach, but you can also use a **mixed model** (LINK TO THE EXTENSIONS CHAPTER) approach, where you model the target as a function of time, but also include a random effect for the grouping variable, in this case, students. This is a very common approach in many areas, and can be very effective. It can also be used for time series data that is not longitudinal, where the random effects are based on autoregressive covariance matrices.

💡 The primary distinguishing feature for referring data as ‘time-series’ and ‘longitudinal’ is the number of time points, where the latter typically has relatively few. This arbitrary though.

### 10.6.1 Time-based Features

When it comes to time-series features, we can apply time-based transformations. One is<sup>12</sup> the **fourier transform**<sup>13</sup>, which can be used to decompose a time series into its component frequencies. This can be useful for identifying periodicity in the data, and can be used as a feature in a model. In marketing contexts, some perform **adstocking** on features, which is a way of modeling the lagged effect of features over time, such that they may have their most important impact immediately, but still can impact the present target from past values. This avoids directly putting lags for each time point as additional features in the model, though that is an option. In that case you have a feature at present time  $t$ , the same feature representing the previous time point  $t-1$ , the feature at  $t-2$ , etc.

Another thing to note about feature transformations, if you have year as a feature, you can use it as a numeric feature or as a categorical feature. In the former case, if you are only considering a linear effect, you should make the

<sup>11</sup>We prefer not to call this **panel data**, though you may also hear of the term in these situations. In our view it's not as informative of a description and exclusively used in econometrics-oriented disciplines.

<sup>12</sup>LINKapproach

<sup>13</sup><https://betterexplained.com/articles/an-interactive-guide-to-the-fourier-transform/>

zero meaningful, typically by starting the year values at zero. This makes the intercept in linear models reference the first year rather than year 0, which can actually make it harder to estimate along with the other coefficients in a model. The same goes if you are using months or days as a numeric feature, since there is no ‘zero’ month. It doesn’t really matter which year/month/day is zero, just that zero refers to one of the actual time points observed.

Dates and/or times can be a bit trickier. Often you can just split dates out into year, month, day, etc., and proceed as discussed. In other cases you’d want to track the day to assess potential seasonal effects, where it makes sense to use a **cyclic** approach (e.g. cyclic spline or sine/cosine transformation<sup>14</sup>) to get at yearly or within-day seasonal effects.

- i** *Weeks are not universal.* Some start on Sunday, others Monday. Some data contexts only consider weekdays. Some systems<sup>a</sup> may have 52 or 53 weeks in a year, and dates may not be in the same week from one year to the next, etc. So use caution when considering weeks as a feature.

<sup>a</sup>[https://en.wikipedia.org/wiki/ISO\\_week\\_date](https://en.wikipedia.org/wiki/ISO_week_date)

---

## 10.7 Spatial Data

TODO: ADD CHAPTER LINKS HERE

We also visit spatial data in a discussion on non-tabular data, but here we want to talk about it from a modeling perspective, especially within the tabular domain. Say you have a target that is a function of location, such as the proportion of people voting a certain way in a county, or the number of crimes in a city. You can use a **spatial regression** approach, where the target is a function of location among other features that may or may not be spatially oriented. Two approaches already discussed may be applied in the case of having continuous spatial features, such as latitude and longitude, or discrete features like county. One model approach to the continuous case include a GAM, where we use a smooth interaction of latitude and longitude. For the discrete setting, we can use a mixed model approach, where we include a random effect for county.

There are other traditional approaches to spatial regression, especially in the continuous spatial domain, such as using a **spatial lag** approach, where we

---

<sup>14</sup><https://developer.nvidia.com/blog/three-approaches-to-encoding-time-information-as-features-for-ml-models/>

incorporate information about the neighborhood of a observation's location into the model. Such models fall under names such as CAR (conditional autoregressive), SAR (spatial autoregressive), BYM, kriging, and so on. These models can be very effective, but are in general different forms of random effects models, and can be seen as special cases of gaussian processes. We feel it's probably unnecessary to get into details of the traditional spatial models unless you know other standard techniques like GAMs or mixed models won't work, or more general approach like gaussian process regression isn't more feasible.

---

## 10.8 Latent Variables

TODO: ADD CHAPTER LINKS HERE

Latent variables are a fundamental aspect of modeling, and simply put, are variables that are not directly observed, but are inferred from other variables. Here are some examples of what might be called latent variables:

- The linear combination of features in a linear regression model is a latent variable, but usually we only think of it as such before the link transformation in GLMs.
- The error term in any model is a latent variable representing all the unknown/unobserved/unmodeled factors that influence the target.
- The principal components in PCA.
- The factor scores in a factor analysis model or structural equation.
- The true target underlying the censored values.
- The clusters in a cluster in cluster analysis.
- The random effects in a mixed model.
- The hidden layers in a deep learning model.

It's easy to see from such a list that latent variables are very common in modeling, so it's good to get comfortable with the concept. Whether they're appropriate to your specific situation will depend on a variety of factors, but they can be very useful in many settings, if not a required part of the modeling approach.

TODO: Measurement error?

## 10.9 Commentary

There's a lot going on with data before you ever get to modeling, and which will affect every aspect of your modeling approach. This chapter outlines common data types, issues, and associated modeling aspects, but in the end, you'll always have to make decisions based on your specific situation, and they will often not be easy. These are only some of the things to consider, so be ready for surprises, and, be ready to learn from them!

---

## 10.10 refs

### Transformations

- min-max vs. standardization [https://sebastianraschka.com/Articles/2014\\_about\\_feature\\_scaling.html](https://sebastianraschka.com/Articles/2014_about_feature_scaling.html)

class imbalance - <https://towardsdatascience.com/handling-imbalanced-datasets-in-machine-learning-7a0e84220f28> - Monroe & Clark - <https://developers.google.com/machine-learning/data-prep/construct/sampling-splitting/imbalanced-data> - <https://machinelearningmastery.com/what-is-imbalanced-classification/>

### calibration

<https://machinelearningmastery.com/probability-calibration-for-imbalanced-classification/> <https://stats.stackexchange.com/questions/452483/why-some-algorithms-produce-calibrated-probabilities>

Niculescu-Mizil, A., & Caruana, R. (2005). Predicting good probabilities with supervised learning. In Proceedings of the 22nd international conference on Machine learning (pp. 625-632).



# 11

---

## *Causal Modeling*

---

All those causal effects will be lost in time, like tears in rain... without adequate counterfactuals

Roy Batty (paraphrased)

Causal inference is a very important topic in machine learning and statistics, and it is also a very difficult topic to understand well, or consistently, because not everyone agrees on how to define **causal** in the first place. Our focus here is merely practical- we just want to show how it plays out in the modeling landscape from a high level overview. This is such a rabbit hole, that we will not be able to go into much detail, but we will try to give you a sense of the landscape, and some of the key ideas.

---

### 11.1 Key ideas

- No model can tell you whether a relationship is causal or not. Causality is inferred, not proven, based on the available evidence.
- The exact same models would be used for similar data settings to answer a causal question, or a predictive question. The difference is in the interpretation of the results.
- Experimental design, such as randomized control trials, are the gold standard for causal inference. But in this case, the gold standard is often not practical, and not without its shortcomings even when it is, and never perfectly implemented. More like a silver standard?
- Causal inference is often done with observational data, which is often the only option, and that's okay.
- Several models exist which are typically employed to answer a more causal-oriented question. These include structural equation models, graphical models, uplift modeling, and more.
- Interactions are the norm, if not the reality. Causal inference generally regards a single effect. If the normal setting is that such an effect would always vary depending on other features, you should question why you want to aggregate your results to a single 'effect', since that effect would be potentially misleading.

## 11.2 Why it matters

Often we need a precise statement about the feature-target relationship, not just whether there is some relationship. For example, we might want to know whether a drug works well, or whether showing an advertisement results in a certain amount of new sales. Whether or not random assignment was used, we generally need to know whether the effect is real, and the size of the effect, and often, the uncertainty in that estimate. Causal modeling is, like machine learning, more of an approach than a specific model, and that approach may involve the design or implementing models we've already seen in a different way to answer the key question. Without more precision in our understanding, we could miss the effect, or overstate it, and make bad decisions as a result.

### 11.2.1 Good to know

Honestly this section is pretty high level, and we are not going to go into much detail here so even just some understanding of correlation and modeling would likely be enough.

---

## 11.3 Classic Experimental Design

Many of those who have taken a statistics course have been exposed to the simple t-test to determine whether two groups are different. While it can be applied to any binary group setting, for our purposes here we can assume the two groups result from some sort of **treatment** that is applied to one group, and not the other. The ‘treatment’ could regard a new drug, demographic groups, a marketing campaign, a new app’s feature, or anything else.

This is a very simple example of an experimental design, and it is a very powerful one. Ideally, we would randomly assign our observational units to the two groups, one which gets the treatment and one which doesn’t. Then we’d measure the difference between the two groups, using some metric to conclude that the two groups are different. This is the basic idea behind the t-test, which would compare the target means of the two groups.

The t-test tells us whether the difference in means between the two groups is *statistically* significant. It definitely does not tell us whether the treatment itself caused the difference, whether the effect is large, nor whether the effect is real, or even if the treatment is a good idea to do in the first place. It just tells us whether the two groups are statistically different.

Turns out, a t-test is just a linear regression. It's a special case of linear regression where there is only one independent variable, and it is a categorical variable with two levels. The coefficient from the linear regression would tell you the mean difference, i.e. as you go from one group to the other, how much does the target mean change? The t-test is just a special case of this, but under the same conditions, the t-statistic from the linear regression and t-test, and corresponding p-value would be identical.

Analysis of variance, or \*\*ANOVA, allows the t-test to be extended to more than two groups, and multiple features, and is also commonly employed to analyze the results of experimental design settings. But ANOVA is still just a linear regression. Even when we get into more complicated design settings such as repeated measures and mixed design, it's still just a linear regression, we'd just be using mixed models. TODO: LINK TO MIXED MODELS

If using a linear regression didn't suggest any notion of causality to you before, it certainly shouldn't now either. The model is *identical* whether there was an experimental design with random assignment or not. The only difference is that the data was collected in a different way, and the theoretical assumptions and motivations are different. Experimental design can give us more confidence in the causal explanation of model results, whatever model is used, and this is why we like to use random assignment when we can. It gives us control for the unobserved factors that might otherwise be influencing the results. If we can be fairly certain the observations are essentially the same *except* for the treatment, then we can be more confident that the treatment is the cause of the difference, and we can be more confident in the causal interpretation of the results. But it doesn't change the model itself, and the results of a model do not in any way prove a causal relationship.

**i** **A/B testing** is just marketing-speak for a two-group setting where one could employ the same mindset they would if they were doing a t-test. It implies randomized assignment, but you'd have to know the context to know if that is actually the case.

---

## 11.4 Natural Experiments

As we noted, random assignment or a formal experiment is not always possible or practical to implement. But sometimes we get to do it anyway, or at least close enough! Sometimes, the world gives us a **natural experiment**, where the assignment to the groups is essentially random, or where there is clear break before and after some event occurs, such that we examine the change as we would in pre-post design.

For example, a certain recent pandemic allowed us to examine vaccination effects, policy effects, remote work, and more. This was not a tightly controlled experiment, but it's something we can treat very similar to an experiment, and we can compare the differences in various outcomes before and after the pandemic to see what changes took place.

## 11.5 Causal Inference

Reasoning about causality is a very old topic, philosophically dating back millenia, and more formally hundreds of years<sup>1</sup>. Random assignment is a relatively new idea, say 150 years old<sup>2</sup>, but was even posited before Wright, Fisher, and Neyman Pearson and the 20th century rise of statistics. But with stats and random assignment we had a way to start using models to help us reason about causal relationships. Pearl and others<sup>3</sup> came along to provide a perspective from computer science, and things have been progressing along. We were actually using programming approaches to do causal inference before back in the 1970s even! Economists eventually got into the game too (e.g., Heckman), though largely reinventing the wheel.

Now we can use recently developed modeling approaches to help us reason about causal relationships, which can be both a blessing and a curse. Our models can be more complex, and we can use more data, which can potentially give us more confidence in our conclusions. But we can still be easily fooled by our models, as well as by ourselves. So we'll need to be careful in how we go about things, but let's see what some of our options are!

## 11.6 Models for Causal Inference

### 11.6.1 Linear Regression

Yep, linear regression<sup>4</sup>. Your old #1 is quite possibly the mostly widely used model for causal inference, historically speaking. We've even already seen linear regression as a graphical model Figure 2.2, and in that sense can serve as the starting point for structural equation models and related, or be used as a baseline for other approaches. Linear regression also tells us for any particular

<sup>1</sup><https://plato.stanford.edu/entries/causation-medieval/>

<sup>2</sup><https://plato.stanford.edu/entries/peirce/>

<sup>3</sup><https://muse.jhu.edu/pub/56/article/867087/summary>

<sup>4</sup><https://matheusfacure.github.io/python-causality-handbook/05-The-Unreasonable-Effectiveness-of-Linear-Regression.html>

effect, what that effect is, accounting for all the other features constant, which kind of already gets into a causal mindset.

However, your standard linear model doesn't care where the data came from and will tell you about group differences whether they come from a randomized experiment or not. And if you don't include features that would have a say in the treatment, you'll potentially get a biased estimate of the effect. As such, linear regression by itself cannot save us from the difficulties of causal inference. But it can be used in conjunction with other approaches, and it can be used to help us reason about causal relationships. For example, we can use it to help us understand the effect of a treatment, or to help us understand the effect of a feature on the target, accounting for other features.

### 11.6.2 Structural Equation Models

TODO: LINK LATENT VARIABLES

**Structural Equation Models** are basically multivariate models, as in multiple targets, used for regression and classification. They are widely employed in the social sciences, and are often used to model both observed and latent variables, with either serving as features or targets. They are also used to model causal relationships, to the point that historically they were called causal graphical models or causal structural models, and are a special case of **graphical models** more generally speaking. They have one of the longest histories of formal statistical modeling dating back over a century<sup>5</sup>. Economists later reinvented the approach under various guises, and computer scientists joined the party after that.

Unfortunately for those looking for causal effects, the basic input for SEM is a correlation matrix, and the basic output is a correlation matrix. Insert your favorite modeling quote here - you know which one. Also, a linear regression and even deep-learning models like autoencoders can be depicted as graphical models, as we have seen. The point is that SEM, like linear regression, can no more tell you whether a relationship is causal than the linear regression, or for that matter, the t-test, could.

### 11.6.3 Counterfactual Thinking

When we think about causality, we really ought to think about **counterfactuals**. What would have happened if I had done something different? What would have happened if I had not done something? What would have happened if I had done something sooner rather than later? What would have happened if I had done nothing at all? These questions are all examples of counterfactual thinking. And this is one of the best ideas to take away from this...

---

<sup>5</sup>Wright<sup>6</sup> is credited with coming up with what would be called **path analysis** in the 1920s, which is a precursor to and part of SEM.

the question is not whether there is a difference between A and B but whether there would still be a difference if A *was* B and B *was* A.

This is the essence of counterfactual thinking. It's not about whether there is a difference between two groups, but whether there would still be a difference if those in one group had actually been treated differently. In this sense, we are concerned with the **potential outcomes** of the treatment, however defined.

Here is a more concrete example:

- Roy is shown ad A, and buys the product.
- Pris is shown ad B, and does not buy the product.

What are we to make of this? Which ad is better? **A** seems to be, but maybe Pris wouldn't have bought the product if shown that ad either, and maybe Roy would have bought the product if shown ad **B** too! With counterfactual thinking, we are concerned with the potential outcomes of the treatment, which in this case is whether or not to show the ad.

Let's say ad A is the new one, i.e., our treatment group, and B is the status quo ad, our control group. Our real question can't be answered by a simple test of whether means or predictions are different among the two groups, as this estimate would be biased if the groups are already different in the first place. The real effect is whether, for those who saw ad A, what the difference in the target would be if they hadn't seen it.

From a prediction stand point, we can get an estimate straightforwardly. For those in the treatment, we can just plug in their feature values with treatment set to ad A. Then we just make a prediction with treatment set to ad B.

#### 11.6.3.1 Python

```
model.predict(X.assign(treatment = 'A')) -
    model.predict(X.assign(treatment = 'B'))
```

#### 11.6.3.2 R

```
predict(model, X |> mutate(treatment = 'A')) -
    predict(model, X |> mutate(treatment = 'B'))
```

With counterfactual thinking explicitly in mind, we can see that the difference in predictions is the difference in the potential outcomes of the treatment.

#### 11.6.4 Uplift Modeling

The counterfactual prediction we just did can be called the **uplift** or **gain** from the treatment. **Uplift modeling** is a general term applied to models where counterfactual thinking is at the forefront, especially in a marketing context. Uplift modeling is not *a* model, but any model that is used to answer

a question about the potential outcomes of a treatment. The key question is what is the gain, or uplift, in applying a treatment vs. not? Typically any statistical model can be used to answer this question, and often the model is a classification model, whether Roy both the product or not.

Some in uplift modeling distinguish:

- **Sure things:** those who would buy the product whether or not shown the ad.
- **Lost causes:** those who would not buy the product whether or not shown the ad.
- **Sleeping dogs:** those who would buy the product if not shown the ad, but not if shown the ad. ‘Do not disturb’!
- **Persuadables:** those who would buy the product if shown the ad, but not if not shown the ad.

One of additional goals in uplift modeling is to identify persuadables for additional marketing efforts, and to avoid wasting money on the lost causes. But to get there, we have to think causally first!

💡 There appear to be more widely used tools for uplift modeling and meta-learners in Python than in R, but there are some options in R as well. In Python you can check out causalml<sup>a</sup> and sci-kit uplift<sup>b</sup> for some nice tutorials and documentation.

<sup>a</sup><https://causalml.readthedocs.io/en/latest/index.html>

<sup>b</sup><https://www.uplift-modeling.com/en/v0.5.1/index.html>

### 11.6.5 Meta-Learning

Meta-learners<sup>7</sup> are used in uplift modeling and other contexts to assess potentially causal relationships between some treatment and outcome.

- **S-learner** - single model for both groups; predict the difference as when all observations are treated vs when all are not, similar to our code demo above.
- **T-learner** - two models, one for each treatment group; predict the difference as when all observations are treated vs when all are not for both models, and take the difference
- **X-learner** - a more complicated modification to the T-learner also using a multi-step approach.
- **Misc-learner** - other meta-learners that are not as popular, but might be applicable for your problem.
- **Transformed outcome:** transform your uplift modeling into a regression

<sup>7</sup><https://arxiv.org/pdf/1706.03461.pdf>

problem in which the prediction is the difference in the potential outcomes. This simplifies the problem to a single model, and can be quite effective.

**i** Meta-learners are not to be confused with **meta-analysis**, which is also related to understanding causal effects. Meta-analysis attempts to combine the results of multiple *studies* to get a better estimate of the true effect. The studies are typically conducted by different researchers and in different settings. **Meta-learning** has also been used to refer to what is more commonly called ensemble learning.

#### 11.6.6 Others approaches

Note that there are many model approaches that would fall under the umbrella of causal inference, and several that are discipline specific but really only a special application of some of the ones we've mentioned here. A few you might come across:

- Marginal structural models
- Instrumental variables and two-stage least squares
- Propensity score matching/weighting
- Meta-analysis
- Bayesian networks

---

#### 11.7 Commentary

You will often hear people speak very strongly about causality in the context of modeling, and those who assume that employing an experimental design solves every problem. But anyone who has actually conducted experiments knows that the implementation is never perfect, often not even close, especially when humans are involved as participants or as experimenters. Experimental design is hard, and if done well, can be very potent, but by itself does not prove anything regarding causality. You will also hear people say that you cannot infer causality from observational data, but it's done all the time, and it's often the only option.

In the end, the main thing is that when we want to make causal statements, we'll make do with what data setting we have, and be careful that we rule out some of the other obvious explanations and issues. The better we can control the setting, or the better we can do things from a model standpoint, the more confident we can be in making causal claims. Causal modeling is an exercise in reasoning, which makes it such an interesting endeavor.

## **11.8 Where to go from here**

We have only scratched the surface here, and there is a lot more to learn. Here are some resources to get you started:

- Barrett, McGowan, and Gerke (2023)
- Cunningham (2023)
- Facure Alves (2022)



# 12

---

## *Misc Models*

---

PLACEHOLDER



# A

---

## References

---

- Barrett, Malcolm, Lucy D'Agostino McGowan, and Travis Gerke. 2023. *Causal Inference in R*. <https://www.r-causal.org/>.
- Brownlee, Jason. 2016. "Gentle Introduction to the Bias-Variance Trade-Off in Machine Learning." *MachineLearningMastery.com*. <https://machinelearningmastery.com/gentle-introduction-to-the-bias-variance-trade-off-in-machine-learning/>.
- . 2021. "Gradient Descent With AdaGrad From Scratch." *MachineLearningMastery.com*. <https://machinelearningmastery.com/gradient-descent-with-adagrad-from-scratch/>.
- Burzykowski, Przemyslaw Biecek and Tomasz. 2020. *Explanatory Model Analysis*. <https://ema.drwhy.ai/>.
- Bycroft, Brendan. n.d. "LLM Visualization." Accessed December 3, 2023. <https://bbycroft.net/llm>.
- Carpenter, Bob. n.d. "Prior Choice Recommendations." *GitHub*. Accessed December 18, 2023. <https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations>.
- causalmkl. 2023. "CausalML." <https://causalmkl.readthedocs.io/en/latest/index.html>.
- Clark, Michael J. 2022. *Generalized Additive Models*. <https://m-clark.github.io/generalized-additive-models/>.
- . 2023. *Mixed Models with R*. <https://m-clark.github.io/mixed-models-with-R/>.
- . n.d.a. *Bayesian Basics*. Accessed December 12, 2023. <https://m-clark.github.io/bayesian-basics/>.
- . n.d.b. *Practical Data Science*. Accessed December 12, 2023. <https://m-clark.github.io/data-processing-and-visualization/>.
- Cohen, Jacob. 2009. *Statistical Power Analysis for the Behavioral Sciences*. 2. ed., reprint. New York, NY: Psychology Press.
- Computing, UCLA Advanced Research. n.d. "FAQ: What Are Pseudo R-Squareds?" Accessed December 11, 2023. <https://stats.oarc.ucla.edu/other/mult-pkg/faq/general/faq-what-are-pseudo-r-squareds/>.
- Cunningham, Scott. 2023. *Causal Inference The Mixtape*. <https://mixtape.scunning.com/>.
- DataBricks. 2019. "What Is AdaGrad?" *Databricks*. <https://www.databricks.com/glossary/adagrad>.
- Facure Alves, Matheus. 2022. "Causal Inference for The Brave and True —

- Causal Inference for the Brave and True.” <https://matheusfacure.github.io/python-causality-handbook/landing-page.html>.
- Fahrmeir, Ludwig, Thomas Kneib, Stefan Lang, and Brian Marx. 2013. *Regression: Models, Methods and Applications*. Berlin, Heidelberg: Springer. <https://doi.org/10.1007/978-3-642-34333-9>.
- Faraway, Julian. 2014. “Linear Models with R.” *Routledge & CRC Press*. <https://www.routledge.com/Linear-Models-with-R/Faraway/p/book/9781439887332>.
- Ferrari, Silvia, and Francisco Cribari-Neto. 2004. “Beta Regression for Modelling Rates and Proportions.” *Journal of Applied Statistics* 31 (7): 799–815. <https://doi.org/10.1080/0266476042000214501>.
- Gelman, Andrew. 2013. “What Are the Key Assumptions of Linear Regression? | Statistical Modeling, Causal Inference, and Social Science.” <https://statmodeling.stat.columbia.edu/2013/08/04/19470/>.
- Gelman, Andrew, John B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin. 2013. *Bayesian Data Analysis, Third Edition*. CRC Press.
- Gelman, Andrew, and Jennifer Hill. 2006. *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Cambridge university press.
- Gelman, Andrew, Jennifer Hill, and Aki Vehtari. 2020. *Regression and Other Stories*. 1st ed. Cambridge University Press. <https://doi.org/10.1017/9781139161879>.
- Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. <https://www.deeplearningbook.org/>.
- Google. n.d.a. “Introduction | Machine Learning.” *Google for Developers*. Accessed December 3, 2023. <https://developers.google.com/machine-learning/decision-forests>.
- . n.d.b. “Machine Learning | Google for Developers.” Accessed December 2, 2023. <https://developers.google.com/machine-learning>.
- Greene, William. 2017. *Econometric Analysis - 8th Edition*. <https://pages.stern.nyu.edu/%7Ewg Greene/Text/econometricanalysis.htm>.
- Grolemund, Hadley Wickham and Garrett. 2023. *Welcome / R for Data Science*. <https://r4ds.hadley.nz/>.
- Harrell, Frank E. 2015. *Regression Modeling Strategies: With Applications to Linear Models, Logistic and Ordinal Regression, and Survival Analysis*. Springer Series in Statistics. Cham: Springer International Publishing. <https://doi.org/10.1007/978-3-319-19425-7>.
- Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. 2017. *Elements of Statistical Learning: Data Mining, Inference, and Prediction. 2nd Edition*. <https://hastie.su.domains/ElemStatLearn/>.
- James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2021. *An Introduction to Statistical Learning*. Vol. 103. Springer Texts in Statistics. New York, NY: Springer New York. <https://doi.org/10.1007/978-1-4614-7138-7>.
- Koenker, Roger. 2000. “Galton, Edgeworth, Frisch, and Prospects for Quan-

- tile Regression in Econometrics.” *Journal of Econometrics* 95 (2): 347–74. [https://doi.org/10.1016/S0304-4076\(99\)00043-3](https://doi.org/10.1016/S0304-4076(99)00043-3).
- . 2005a. *Quantile Regression*. Vol. 38. Cambridge university press. [https://books.google.com/books?hl=en&lr=&id=WjOdAgAAQBAJ&oi=fnd&pg=PT12&dq=info:E32s5Y3j4NMJ:scholar.google.com&ots=CQFHSt5o-W&sig=G1TpKPHo-BRdJ8qWcBrIBI2FQAs](https://books.google.com/books?hl=en&lr=&id=WjOdAgAAQBAJ&oi=fnd&pg=PT12&dq=koenker+quantile+regression&ots=CQFHSt5o-W&sig=G1TpKPHo-BRdJ8qWcBrIBI2FQAs).
- . 2005b. *Quantile Regression*. Vol. 38. Cambridge university press. <https://books.google.com/books?hl=en&lr=&id=WjOdAgAAQBAJ&oi=fnd&pg=PT12&dq=info:E32s5Y3j4NMJ:scholar.google.com&ots=CQFHSt5qY&sig=E8zXmNabc0nGmYJlSYjQqhzyVT8>.
- Kruschke, John. 2010. *Doing Bayesian Data Analysis: A Tutorial Introduction with R*. Academic Press.
- Kuhn, Max, and Julia Silge. 2023. *Tidy Modeling with R*. <https://www.tidyverse.org/>.
- Künzel, Sören R., Jasjeet S. Sekhon, Peter J. Bickel, and Bin Yu. 2019. “Metalearners for Estimating Heterogeneous Treatment Effects Using Machine Learning.” *Proceedings of the National Academy of Sciences* 116 (10): 4156–65. <https://doi.org/10.1073/pnas.1804597116>.
- Lang, Michel, Martin Binder, Jakob Richter, Patrick Schratz, Florian Pfisterer, Stefan Coors, Quay Au, Giuseppe Casalicchio, Lars Kotthoff, and Bernd Bischl. 2019. “Mlr3: A Modern Object-Oriented Machine Learning Framework in R.” *Journal of Open Source Software* 4 (44): 1903. <https://doi.org/10.21105/joss.01903>.
- Masis, Serg. 2023. “Interpretable Machine Learning with Python - Second Edition.” *Packt*. <https://www.packtpub.com/product/interpretable-machine-learning-with-python-second-edition/9781803235424>.
- McElreath, Richard. 2020. “Statistical Rethinking: A Bayesian Course with Examples in R and STAN.” *Routledge & CRC Press*. <https://www.routledge.com/Statistical-Rethinking-A-Bayesian-Course-with-Examples-in-R-and-STAN/McElreath/p/book/9780367139919>.
- . n.d. “Statistical Rethinking: A Bayesian Course with Examples in R and STAN.” *Routledge & CRC Press*. Accessed December 10, 2023. <https://www.routledge.com/Statistical-Rethinking-A-Bayesian-Course-with-Examples-in-R-and-STAN/McElreath/p/book/9780367139919>.
- Molnar, Christoph. 2023. *Interpretable Machine Learning*. <https://christophm.github.io/interpretable-ml-book/>.
- Morgan, Stephen, and Christopher Winship. 2014. “Counterfactuals and Causal Inference: Methods and Principles for Social Research, 2nd Edition,” January. <https://stars.library.ucf.edu/etextbooks/298>.
- Murphy, Kevin P. 2012. “Machine Learning: A Probabilistic Perspective.” *MIT Press*. <https://mitpress.mit.edu/9780262018029/machine-learning/>.
- . 2023. “Probabilistic Machine Learning.” *MIT Press*. <https://mitpress.mit.edu/9780262046824/probabilistic-machine-learning/>.
- Pearl, Judea. 2009. “Causal Inference in Statistics: An Overview.” *Statistics Surveys* 3 (none): 96–146. <https://doi.org/10.1214/09-SS057>.

- . n.d. “Causal Inference: History, Perspectives, Adventures, and Unification (An Interview with Judea Pearl).” Accessed December 10, 2023. <https://muse.jhu.edu/pub/56/article/867087/summary>.
- Pok, Wilson. n.d. “How Uplift Modeling Works | Blogs.” Accessed December 10, 2023. <https://ambianta.com/blog/2020-07-07-uplift-modeling/>.
- Roback, Paul, and Julie Legler. 2021. *Beyond Multiple Linear Regression*. <https://bookdown.org/robback/bookdown-BeyondMLR/>.
- Rovine, Michael J, and Douglas R Anderson. 2004. “Peirce and Bowditch.” *The American Statistician* 58 (3): 232–36. <https://doi.org/10.1198/000313004X964>.
- Schmidhuber, Juergen. 2022. “Annotated History of Modern AI and Deep Learning.” arXiv. <https://doi.org/10.48550/arXiv.2212.11279>.
- scikit-learn. 2023. “1.16. Probability Calibration.” *Scikit-Learn*. <https://scikit-learn/stable/modules/calibration.html>.
- . n.d. “Nested Versus Non-Nested Cross-Validation.” *Scikit-Learn*. Accessed December 3, 2023. [https://scikit-learn/stable/auto\\_examples/model\\_selection/plot\\_nested\\_cross\\_validation\\_iris.html](https://scikit-learn/stable/auto_examples/model_selection/plot_nested_cross_validation_iris.html).
- Shevchenko, Maksim. n.d. “Types of Customers — Scikit-Uplift 0.3.1 Documentation.” Accessed December 10, 2023. [https://www.uplift-modeling.com/en/v0.5.1/user\\_guide/introduction/clients.html](https://www.uplift-modeling.com/en/v0.5.1/user_guide/introduction/clients.html).
- StatQuest with Josh Starmer. 2019a. “Gradient Descent, Step-by-Step.” <https://www.youtube.com/watch?v=sDv4f4s2SB8>.
- . 2019b. “Stochastic Gradient Descent, Clearly Explained!!!” <https://www.youtube.com/watch?v=vMh0zPT0tLI>.
- VanderPlas, Jake. 2016. “Python Data Science Handbook [Book].” <https://www.oreilly.com/library/view/python-data-science/9781491912126/>.
- Welchowski, Thomas, Kelly O. Maloney, Richard Mitchell, and Matthias Schmid. 2022. “Techniques to Improve Ecological Interpretability of Black-Box Machine Learning Models.” *Journal of Agricultural, Biological and Environmental Statistics* 27 (1): 175–97. <https://doi.org/10.1007/s13253-021-00479-7>.
- Wikipedia. 2023. “Relationships Among Probability Distributions.” *Wikipedia*. [https://en.wikipedia.org/w/index.php?title=Relationships\\_among\\_probability\\_distributions&oldid=1180084573](https://en.wikipedia.org/w/index.php?title=Relationships_among_probability_distributions&oldid=1180084573).
- Wood, Simon N. 2017. *Generalized Additive Models: An Introduction with R, Second Edition*. 2nd ed. Boca Raton: Chapman; Hall/CRC. <https://doi.org/10.1201/9781315370279>.
- Wooldridge, Jeffrey M. 2012. *Introductory Econometrics: A Modern Approach*. 5th edition. Mason, OH: Cengage Learning.
- Zhang, Aston, Zack Lipton, Mu Li, and Alex Smola. n.d. “Dive into Deep Learning — Dive into Deep Learning 1.0.3 Documentation.” Accessed December 10, 2023. <https://d2l.ai/index.html>.

# B

---

## *Dataset Descriptions*

---

### B.1 Heart Failure

Dataset from Davide Chicco, Giuseppe Jurman: "Machine learning can predict survival of patients with heart failure from serum creatinine and ejection fraction alone. BMC Medical Informatics and Decision Making 20, 16 (2020)

<https://www.kaggle.com/datasets/andrewmvd/heart-failure-clinical-data>

Cardiovascular diseases (CVDs) are the number 1 cause of death globally, taking an estimated 17.9 million lives each year, which accounts for 31% of all deaths worldwide. Heart failure is a common event caused by CVDs and this dataset contains 12 features that can be used to predict mortality by heart failure.

Most cardiovascular diseases can be prevented by addressing behavioural risk factors such as tobacco use, unhealthy diet and obesity, physical inactivity and harmful use of alcohol using population-wide strategies.

People with cardiovascular disease or who are at high cardiovascular risk (due to the presence of one or more risk factors such as hypertension, diabetes, hyperlipidaemia or already established disease) need early detection and management wherein a machine learning model can be of great help.

age: Age

anaemia: Decrease of red blood cells or hemoglobin (boolean)

creatinine\_phosphokinase: Level of the CPK enzyme in the blood (mcg/L)

diabetes: If the patient has diabetes (boolean)

ejection\_fraction: Percentage of blood leaving the heart at each contraction (percentage)

high\_blood\_pressure: If the patient has hypertension (boolean)

platelets: Platelets in the blood (kiloplatelets/mL)

serum\_creatinine: Level of serum creatinine in the blood (mg/dL)

serum\_sodium: Level of serum sodium in the blood (mEq/L)

sex: Woman or man (binary)

smoking: If the patient smokes or not (boolean)

time: Follow-up period (days)

DEATH\_EVENT: If the patient deceased during the follow-up period (boolean)

For booleans: Sex - Gender of patient Male = 1, Female =0 (renamed for our data) Age - Age of patient Diabetes - 0 = No, 1 = Yes Anaemia - 0 = No, 1 = Yes High\_blood\_pressure - 0 = No, 1 = Yes Smoking - 0 = No, 1 = Yes DEATH\_EVENT - 0 = No, 1 = Yes

## B.2 Heart Disease UCI

Dataset from Kaggle: <https://www.kaggle.com/ronitf/heart-disease-uci>

This database contains 76 attributes, but all published experiments refer to using a subset of 14 of them. In particular, the Cleveland database is the only one that has been used by ML researchers to this date.

Attribute Information:

name	role	type	demographic	description	units	missing_values
0	age	Feature	Integer	Age	None	1
1	sex	Feature	Categorical	Sex	None	0
2	cp	Feature	Categorical	Resting	blood pressure	0
3	trestbps	Feature	Integer	(on admission to the hospital)	mm Hg	0
4	chol	Feature	Integer	serum cholesterol	mg/dl	0
5	fbs	Feature	Categorical	Fasting	blood sugar	> 120 mg/dl
6	restecg	Feature	Categorical	Exercise	maximum heart rate	achieved
7	thalach	Feature	Integer	thalassemia	ST depression	induced by exercise
8	exang	Feature	Categorical	Exercise	induced	angina
9	oldpeak	Feature	Integer	thalassemia	ST depression	relative to rest
10	slope	Feature	Categorical	thalassemia	induced by exercise	... (0-3)
11	ca	Feature	Integer	number of major vessels	colored by flour...	0-3
12	thal	Feature	Categorical	thalassemia	yes	0-3
13	num	Target	Integer	diagnosis of heart disease	None	0-1

Features and target were renamed for our data.

# C

---

## *Matrix Operations*

---

Addition, subtraction, and multiplication. These are all things you already know how to do with scalars. What happens, though, if you want to multiply two different matrices together. Does that simple, scalar operation still translate if you have a  $2 \times 3$  matrix and a  $3 \times 2$  matrix? If you said, “Yes!”, or if words like matrix and scalar make you break out in a sweat, then this chapter is for you! Maybe you’ve encountered these concepts before, possibly in the first 3 weeks of a graduate statistics course; you left that class confused, angry, and wondering why you would be subjected to such nonsense. If that sounds familiar, this chapter is for you. If you found Linear Algebra easy, then you can comfortably skip this chapter and see if Chapter 2 might be where you want to start.

Matrix operations, especially multiplication, are critical for understanding what comes throughout the rest of the book. Knowing the underlying mechanics of matrix operations helps to demystify several issues that you might run into with your models. You’ll frequently see model output tell you how many records were deleted due to missingness. You’ll then find yourself wondering why is missingness such a problem that entire rows get deleted from your model? As we progress through the next several chapters, those issues and more will become much clearer.

Before we get into any operations, though, let’s make sure we are together on some concepts.

A *scalar* is a single value. It might help if you think about a scalar as a single block @ref(fig:numbered\_block)

```
scalar_example = 1  
scalar_example = 1
```

Just like we can line blocks up on the floor, we can put our scalars together to form a *vector* @ref(fig:vector\_blocks). A vector is a collection of scalars with a length of **n**.

```
vector_example = 1:5
```

This vector containing values from 1 to 5 would have a length of 5.

```
vector_example = range(0, 5)
```

Now, we can take a few of our block vectors and assemble them into a *matrix*. A matrix is a 2 dimensional collection of vectors.

@ref{fig:matrix\_blocks}

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

If you think about most tables you've ever seen, you'll see that the simple matrix looks remarkably familiar!

A matrix has 2 dimensions, rows and columns. When we talk about the dimensions of a matrix, we always make note of the rows first, followed by the columns. This matrix has 2 rows and 3 columns; therefore, we have a  $2 \times 3$  matrix.

## C.1 Addition

Matrix addition, along with subtraction, is the easiest concept when dealing with matrices. While it is easy to grasp, you will not find it featured as prominently as matrix multiplication.

There is one rule for matrix addition: the matrices need to have the same dimensions.

Let's check out these two matrices:

$$\begin{array}{c} \text{Matrix A} \\ \begin{bmatrix} 1_{11} & 2_{12} & 3_{13} \\ 4_{21} & 5_{22} & 6_{23} \end{bmatrix} \end{array} \quad \begin{array}{c} \text{Matrix B} \\ \begin{bmatrix} 7_{11} & 8_{12} & 9_{13} \\ 9_{21} & 8_{22} & 7_{23} \end{bmatrix} \end{array}$$

You probably noticed that we gave each scalar within the matrix a label associated with its row and column position. We can use these to see how we will produce the new matrix:

Now, we can set this up as an addition problem to produce Matrix C:

$$\begin{array}{c} \text{Matrix A} \\ \begin{bmatrix} 1_{11} & 2_{12} & 3_{13} \\ 4_{21} & 5_{22} & 6_{23} \end{bmatrix} \end{array} + \begin{array}{c} \text{Matrix B} \\ \begin{bmatrix} 7_{11} & 8_{12} & 9_{13} \\ 9_{21} & 8_{22} & 7_{23} \end{bmatrix} \end{array} = \begin{array}{c} \text{Matrix C} \\ \begin{bmatrix} A_{11} + B_{11} & A_{12} + B_{12} & A_{13} + B_{13} \\ A_{21} + B_{21} & A_{22} + B_{22} & A_{23} + B_{23} \end{bmatrix} \end{array}$$

Now we can pull in the real numbers:

$$\begin{array}{c} \text{Matrix A} \\ \left[ \begin{array}{ccc} 1_{11} & 2_{12} & 3_{13} \\ 4_{21} & 5_{22} & 6_{23} \end{array} \right] \end{array} + \begin{array}{c} \text{Matrix B} \\ \left[ \begin{array}{ccc} 7_{11} & 8_{12} & 9_{13} \\ 9_{21} & 8_{22} & 7_{23} \end{array} \right] \end{array} = \begin{array}{c} \text{Matrix C} \\ \left[ \begin{array}{ccc} 1+7 & 2+8 & 3+9 \\ 4+9 & 5+8 & 6+7 \end{array} \right] \end{array}$$

Giving us Matrix C:

$$\begin{array}{c} \text{Matrix A} \\ \left[ \begin{array}{ccc} 1_{11} & 2_{12} & 3_{13} \\ 4_{21} & 5_{22} & 6_{23} \end{array} \right] \end{array} + \begin{array}{c} \text{Matrix B} \\ \left[ \begin{array}{ccc} 7_{11} & 8_{12} & 9_{13} \\ 9_{21} & 8_{22} & 7_{23} \end{array} \right] \end{array} = \begin{array}{c} \text{Matrix C} \\ \left[ \begin{array}{ccc} 8 & 10 & 12 \\ 13 & 13 & 13 \end{array} \right] \end{array}$$


---

## C.2 Subtraction

Take everything that you just saw with addition and replace it with subtraction!

Just like addition, every matrix needs to have the same dimensions if you are going to use subtraction.

Let's see those two matrices again and cast it as subtraction problem:

$$\begin{array}{c} \text{Matrix A} \\ \left[ \begin{array}{ccc} 1_{11} & 2_{12} & 3_{13} \\ 4_{21} & 5_{22} & 6_{23} \end{array} \right] \end{array} - \begin{array}{c} \text{Matrix B} \\ \left[ \begin{array}{ccc} 7_{11} & 8_{12} & 9_{13} \\ 9_{21} & 8_{22} & 7_{23} \end{array} \right] \end{array} = \begin{array}{c} \text{Matrix C} \\ \left[ \begin{array}{ccc} A_{11} - B_{11} & A_{12} - B_{12} & A_{13} - B_{13} \\ A_{21} - B_{21} & A_{22} - B_{22} & A_{23} - B_{23} \end{array} \right] \end{array}$$

And now we can substitute in the real numbers:

$$\begin{array}{c} \text{Matrix A} \\ \left[ \begin{array}{ccc} 1_{11} & 2_{12} & 3_{13} \\ 4_{21} & 5_{22} & 6_{23} \end{array} \right] \end{array} - \begin{array}{c} \text{Matrix B} \\ \left[ \begin{array}{ccc} 7_{11} & 8_{12} & 9_{13} \\ 9_{21} & 8_{22} & 7_{23} \end{array} \right] \end{array} = \begin{array}{c} \text{Matrix C} \\ \left[ \begin{array}{ccc} 1-7 & 2-8 & 3-9 \\ 4-9 & 5-8 & 6-7 \end{array} \right] \end{array}$$

And end with this matrix:

$$\begin{array}{c} \text{Matrix A} \\ \left[ \begin{array}{ccc} 1_{11} & 2_{12} & 3_{13} \\ 4_{21} & 5_{22} & 6_{23} \end{array} \right] \end{array} - \begin{array}{c} \text{Matrix B} \\ \left[ \begin{array}{ccc} 7_{11} & 8_{12} & 9_{13} \\ 9_{21} & 8_{22} & 7_{23} \end{array} \right] \end{array} = \begin{array}{c} \text{Matrix C} \\ \left[ \begin{array}{ccc} -6 & -6 & -6 \\ -5 & -3 & -1 \end{array} \right] \end{array}$$

Adding and subtracting matrices in R and Python is pretty simple.

In R, we can create a matrix a few ways: with the matrix function or by row binding numeric vectors.

```

matrix_A = rbind(1:3,
                 4:6)

# The following is an equivalent
# to rbind:
# matrix_A = matrix(c(1:3, 4:6),
#                     nrow = 2,
#                     ncol = 3, byrow = TRUE)

matrix_B = rbind(7:9,
                 9:7)

```

Once we have those matrices created, we can use the standard + and - signs to add and subtract:

```

matrix_A + matrix_B

[,1] [,2] [,3]
[1,]    8   10   12
[2,]   13   13   13

matrix_A - matrix_B

[,1] [,2] [,3]
[1,]   -6   -6   -6
[2,]   -5   -3   -1

```

The task is just as easy in Python. We will import `numpy` and then use the `matrix` method to create the matrices:

```

import numpy as np

matrix_A = np.matrix('1 2 3; 4 5 6')

matrix_B = np.matrix('7 8 9; 9 8 7')

```

Just like R, we can use + and - on those matrices.

```

matrix_A + matrix_B

matrix([[ 8, 10, 12],
       [13, 13, 13]])

matrix_A - matrix_B

matrix([[-6, -6, -6],
       [-5, -3, -1]])

```

### C.3 Transpose

As you progress through this book, you might see a matrix denoted as  $A^T$ ; here the superscripted T stands for *transpose*. If we transpose a matrix, all we are doing is flipping the rows and columns along the matrix's main diagonal. A visual example is much easier:

$$\begin{array}{c} \text{Matrix A} \\ \left[ \begin{matrix} 1_{11} & 2_{12} & 3_{13} \\ 4_{21} & 5_{22} & 6_{23} \end{matrix} \right] \end{array} \rightarrow \begin{array}{c} \text{Matrix } A^T \\ \left[ \begin{matrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{matrix} \right] \end{array}$$

Like any matrix operation, a transpose is pretty easy to do when the matrix is small; you're best bet is to rely on software to do anything beyond a few rows or columns.

In R, all we need is the `t` function:

```
t(matrix_A)

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

In Python, we can use numpy's `transpose` method:

```
matrix_A.transpose()

matrix([[1, 4],
       [2, 5],
       [3, 6]])
```

### C.4 Multiplication

Now, you probably have some confidence in doing matrix operations. Just as quickly as we built that confidence, it will be crushed when learning about matrix multiplication.

When dealing with matrix multiplication, we have a huge change to our rule. No longer can our dimensions be the same! Instead, the matrices need to be *conformable* – the first matrix needs to have the same number of columns

as the number of rows within the second matrix. In other words, the inner dimensions must match.

Look one more time at these matrices:

$$\begin{matrix} \text{Matrix A} \\ \begin{bmatrix} 1_{11} & 2_{12} & 3_{13} \\ 4_{21} & 5_{22} & 6_{23} \end{bmatrix} \end{matrix} \cdot \begin{matrix} \text{Matrix B} \\ \begin{bmatrix} 7_{11} & 8_{12} & 9_{13} \\ 9_{21} & 8_{22} & 7_{23} \end{bmatrix} \end{matrix}$$

Matrix A has dimensions of  $2x3$ , as does Matrix B. Putting those dimensions side by side –  $2x3 * 2x3$  – we see that our inner dimensions are 3 and 2 and do not match.

What if we *transpose* Matrix B?

$$\begin{matrix} \text{Matrix } B^T \\ \begin{bmatrix} 7_{11} & 9_{12} \\ 8_{21} & 8_{22} \\ 9_{31} & 7_{32} \end{bmatrix} \end{matrix}$$

Now we have something that works!

$$\begin{matrix} \text{Matrix A} \\ \begin{bmatrix} 1_{11} & 2_{12} & 3_{13} \\ 4_{21} & 5_{22} & 6_{23} \end{bmatrix} \end{matrix} \cdot \begin{matrix} \text{Matrix } B^T \\ \begin{bmatrix} 7_{11} & 9_{12} \\ 8_{21} & 8_{22} \\ 9_{31} & 7_{32} \end{bmatrix} \end{matrix} = \begin{matrix} \text{Matrix C} \\ \begin{bmatrix} \cdot & \cdot \\ \cdot & \cdot \end{bmatrix} \end{matrix}$$

Now we have a  $2x3 * 3x2$  matrix multiplication problem! The resulting matrix will have the same dimensions as our two matrices' outer dimensions:  $2x2$

Here is how we will get at  $2x2$  matrix:

$$\begin{matrix} \text{Matrix A} \\ \begin{bmatrix} 1_{11} & 2_{12} & 3_{13} \\ 4_{21} & 5_{22} & 6_{23} \end{bmatrix} \end{matrix} \cdot \begin{matrix} \text{Matrix } B^T \\ \begin{bmatrix} 7_{11} & 9_{12} \\ 8_{21} & 8_{22} \\ 9_{31} & 7_{32} \end{bmatrix} \end{matrix} = \begin{matrix} \text{Matrix C} \\ \begin{bmatrix} (A_{11} * B_{11}) + (A_{12} * B_{21}) + (A_{13} * B_{31}) & (A_{11} * B_{12}) + (A_{12} * B_{22}) + (A_{13} * B_{32}) \\ (A_{21} * B_{11}) + (A_{22} * B_{21}) + (A_{23} * B_{31}) & (A_{21} * B_{12}) + (A_{22} * B_{22}) + (A_{23} * B_{32}) \end{bmatrix} \end{matrix}$$

That might look like a horrible mess and likely isn't easy to commit to memory. Instead, we'd like to show you a way that might make it easier to remember how to multiply matrices. It also gives a nice representation of why your matrices need to be conformable.

We can leave Matrix A exactly where it is, flip Matrix  $B^T$ , and stack it right on top of Matrix A:

$$\begin{bmatrix} 9_b & 8_b & 7_b \\ 7_b & 8_b & 9_b \\ 1_a & 2_a & 3_a \\ 4_a & 5_a & 6_a \end{bmatrix}$$

Now, we can let those rearranged columns from Matrix  $B^T$  “fall down” through the rows of Matrix A:

$$\begin{bmatrix} 9_b & 8_b & 7_b \\ 1_a * 7_b & 2_a * 8_b & 3_a * 9_b \\ 4_a & 5_a & 6_a \end{bmatrix} = \begin{bmatrix} 50 & \cdot \\ \cdot & \cdot \end{bmatrix} \text{ Matrix C}$$

Adding those products together gives us 50 for  $C_{11}$ .

Let’s move that row down to the next row in the Matrix A, multiply, and sum the products.

$$\begin{bmatrix} 9_b & 8_b & 7_b \\ 1_a & 2_a & 3_a \\ 4_a * 7_b & 5_a * 8_b & 6_a * 9_b \end{bmatrix} = \begin{bmatrix} 50 & \cdot \\ 122 & \cdot \end{bmatrix} \text{ Matrix C}$$

We have 122 for  $C_{21}$ . That first column from Matrix  $B^T$  won’t be used any more, but now we need to move the second column through Matrix A.

$$\begin{bmatrix} 1_a * 9_b & 2_a * 8_b & 3_a * 7_b \\ 4_a & 5_a & 6_a \end{bmatrix} = \begin{bmatrix} 50 & 46 \\ 122 & \cdot \end{bmatrix} \text{ Matrix C}$$

That gives us 46 for  $C_{12}$ .

And finally:

$$\begin{bmatrix} 1_a & 2_a & 3_a \\ 4_a * 9_b & 5_a * 8_b & 6_a * 7_b \end{bmatrix} = \begin{bmatrix} 50 & 46 \\ 122 & 118 \end{bmatrix} \text{ Matrix C}$$

We have 118 for  $C_{22}$ .

Now that you know how these work, you can see how easy it is to handle these tasks in R and Python.

In R, we need to use a fancy operator: `%*%`. This is just R's matrix multiplication operator. We will also use the transpose function: `t`.

```
matrix_A %*% t(matrix_B)
[,1] [,2]
[1,] 50   46
[2,] 122  118
```

In Python, we can just use the regular multiplication operator and the transpose method:

```
matrix_A * matrix_B.transpose()
matrix([[ 50,  46],
       [122, 118]])
```

You can see that whether we do this by hand, R, or Python, we come up with the same answer! While these small matrices can definitely be done by hand, we will always trust the computer to handle larger matrices.

## C.5 Inversion

You might want to think of *matrix inversion* as the reciprocal of the matrix, usually noted as  $A^{-1}$ . The biggest reason that we might invert a matrix is because there is no matrix division.

Inversion can only be performed on square matrices (e.g.,  $2 \times 2$ ,  $3 \times 3$ ,  $4 \times 4$ ) and the *determinant* of a matrix cannot be 0. Since the determinant is important for finding the inverse, we should probably have an idea about how to find the determinant.

### C.5.1 Matrix Determinant

While we've been using the matrix row/column positions in our examples, we are going to shift to letters to label the positions. We can start with a  $2 \times 2$  matrix:

$$\begin{array}{c} \text{Matrix C} \\ \left[ \begin{array}{cc} A & B \\ C & D \end{array} \right] \end{array}$$

To find the determinant, we would take  $|C| = (A * D) - (B * C)$ .

Returning back to Matrix C, we have  $|C| = (50_a * 118_d) - (46_b * 122_c) = 288$

$$\begin{array}{c} \text{Matrix C} \\ \left[ \begin{array}{cc} 50 & 46 \\ 122 & 118 \end{array} \right] \end{array}$$

A  $3 \times 3$  matrix doesn't pose much more of a challenge.

$$\begin{array}{c} \text{Matrix D} \\ \left[ \begin{array}{ccc} A & B & C \\ D & E & F \\ G & H & I \end{array} \right] \end{array}$$

The canonical form might not be as intuitive, but it is worth seeing:

$$|D| = A \begin{vmatrix} E & I \\ F & H \end{vmatrix} - B \begin{vmatrix} D & I \\ F & G \end{vmatrix} + C \begin{vmatrix} D & H \\ E & G \end{vmatrix}$$

Breaking it down a bit further will help to see where all of the values go:

$$|D| = A(E * I - F * H) - B(D * I - F * G) + C(D * H - E * G)$$

Now we can work that out with a real matrix:

$$\begin{array}{c} \text{Matrix D} \\ \left[ \begin{array}{ccc} 2 & 1 & 3 \\ 6 & 5 & 4 \\ 7 & 8 & 9 \end{array} \right] \end{array}$$

To get our determinant:

$$|D| = 2(5 * 9 - 4 * 8) - 1(6 * 9 - 4 * 7) + 3(6 * 8 - 5 * 7) = 39$$

And just to confirm that our math is correct, we can check for the determinant in R and Python.

R has a handy function called `det`:

```
matrix_D = matrix(c(2, 1, 3,
                   6, 5, 4,
                   7, 8, 9),
                   nrow = 3,
                   ncol = 3,
                   byrow = TRUE)

det(matrix_D)
```

```
[1] 39
```

We can keep using `numpy`, but we will have to use `det` within the `linalg` module.

```
matrix_D = np.matrix('2 1 3; 6 5 4; 7 8 9')  
  
np.linalg.det(matrix_D)  
  
38.99999999999999
```

Just to show you how this pattern would continue

You can find a lot of examples online on how to do  $2 \times 2$  and  $3 \times 3$  matrix inversions, mostly because they are the easiest to do.

How do you know that you properly inverted your matrix? You multiply the original matrix by the inverse matrix and you will get an *identity* matrix.

We have a nice figure in Figure @ref(fig:hello), and also a table in Table @ref(tab:iris).

# D

---

## *Other to come*

---

---

### D.1 Simulation

---

### D.2 Bayesian Demonstration

Metropolis-Hastings demo

```
# Define the log-likelihood function for linear regression
log_likelihood <- function(beta, X, y, sigma_sq) {
  y_hat <- X %*% beta
  residuals <- y - y_hat
  log_likelihood <- -0.5 * length(y) * log(2 * pi * sigma_sq) - 0.5 * sum(residuals^2) / sigma_sq
  return(log_likelihood)
}

# Define the prior distribution for beta
prior_beta <- function(beta) {
  prior_mean <- rep(0, length(beta))
  prior_sd <- rep(10, length(beta))
  log_prior <- sum(dnorm(beta, mean = prior_mean, sd = prior_sd, log = TRUE))
  return(log_prior)
}

# Define the prior distribution for sigma
prior_sigma <- function(sigma_sq) {
  alpha <- 2
  beta <- 2
  # log_prior <- dgamma(1/sigma_sq, shape = alpha, rate = beta, log = TRUE)
  log_prior <- extraDistr::dinvgamma(sigma_sq, alpha = alpha, beta = beta, log = TRUE)

  return(log_prior)
}

# Define the proposal distribution for beta
```

```

proposal_beta <- function(beta, scale) {
  beta_proposal <- rnorm(length(beta), mean = beta, sd = scale)
  return(beta_proposal)
}

# Define the proposal distribution for sigma
proposal_sigma <- function(sigma_sq, scale) {
  # sigma_proposal <- rgamma(1, shape = sigma_sq / scale, rate = scale)
  sigma_proposal <- extraDistr::rinvgamma(1, alpha = sigma_sq / scale, beta = scale)
  return(sigma_proposal)
}

# Set up the data
# set.seed(123)
# n <- 100
# X <- cbind(1, rnorm(n), rnorm(n), rnorm(n))
# beta_true <- c(1, 2, 3, 4)/4
# sigma_true <- 1
# y <- X %*% beta_true + rnorm(n, sd = sigma_true)

# Set up the Metropolis-Hastings algorithm
# n_iter <- 10000

# Run the Metropolis-Hastings algorithm
mh = function(
  X,
  y,
  beta = rep(0, ncol(X)),
  sigma_sq = .5,
  scale_beta = 0.1,
  scale_sigma = 1,
  chains = 2,
  warmup = 1000,
  n_iter = 2000,
  seed = 123
) {
  set.seed(seed)

  result <- list()
  beta_start <- beta
  sigma_sq_start <- sigma_sq

  for (c in 1:chains){
    acceptance_beta <- 0

```

```

acceptance_sigma <- 0
beta_samples <- matrix(0, n_iter, ncol(X))
sigma_sq_samples <- rep(0, n_iter)

if (c > 1) {
  beta <- beta_start
  sigma_sq <- sigma_sq_start
}

for (i in 1:n_iter) {
  # Update beta
  beta_proposal <- proposal_beta(beta, scale_beta)
  log_ratio_beta <- log_likelihood(beta_proposal, X, y, sigma_sq) + prior_beta(beta_proposal) -
    log_likelihood(beta, X, y, sigma_sq) - prior_beta(beta)
  if (log(runif(1)) < log_ratio_beta) {
    beta <- beta_proposal
    acceptance_beta <- acceptance_beta + 1
  }
  beta_samples[i, ] <- beta

  # Update sigma_sq
  sigma_sq_proposal <- proposal_sigma(sigma_sq, scale_sigma)
  log_ratio_sigma <- log_likelihood(beta, X, y, sigma_sq_proposal) + prior_sigma(sigma_sq_proposal) -
    log_likelihood(beta, X, y, sigma_sq) - prior_sigma(sigma_sq)
  if (log(runif(1)) < log_ratio_sigma) {
    sigma_sq <- sigma_sq_proposal
    acceptance_sigma <- acceptance_sigma + 1
  }
  sigma_sq_samples[i] <- sigma_sq
}

message("Acceptance rate for beta:", acceptance_beta / n_iter, "\n")
message("Acceptance rate for sigma:", acceptance_sigma / n_iter, "\n")

result[[c]] = list(
  beta = beta_samples[-(1:warmup), ],
  sigma_sq = sigma_sq_samples[-(1:warmup)],
  # y_rep = X %*% t(beta_samples[-(1:warmup), ])
  # +rnorm(n_iter - warmup, sd = sqrt(sigma_sq_samples[-(1:warmup)]))
  y_rep = t(X %*% t(beta_samples[-(1:warmup), ]) + rnorm(n_iter - warmup, sd = sqrt(sigma_sq_samples[-(1:warmup)])))
)
}
result
}

```

```
X_train = df_happiness |>
  select(life_exp, gdp_pc, corrupt) |>
  as.matrix()

our_result = mh(
  X = cbind(1, X_train),
  y = df_happiness$happiness,
  beta = c(mean(df_happiness$happiness), rep(0, ncol(X_train))),
  sigma_sq = var(df_happiness$happiness),
  scale_sigma = .5,
  warmup = 1000,
  n_iter = 2000
)

str(our_result)
```

---

### D.3 Linear Programming