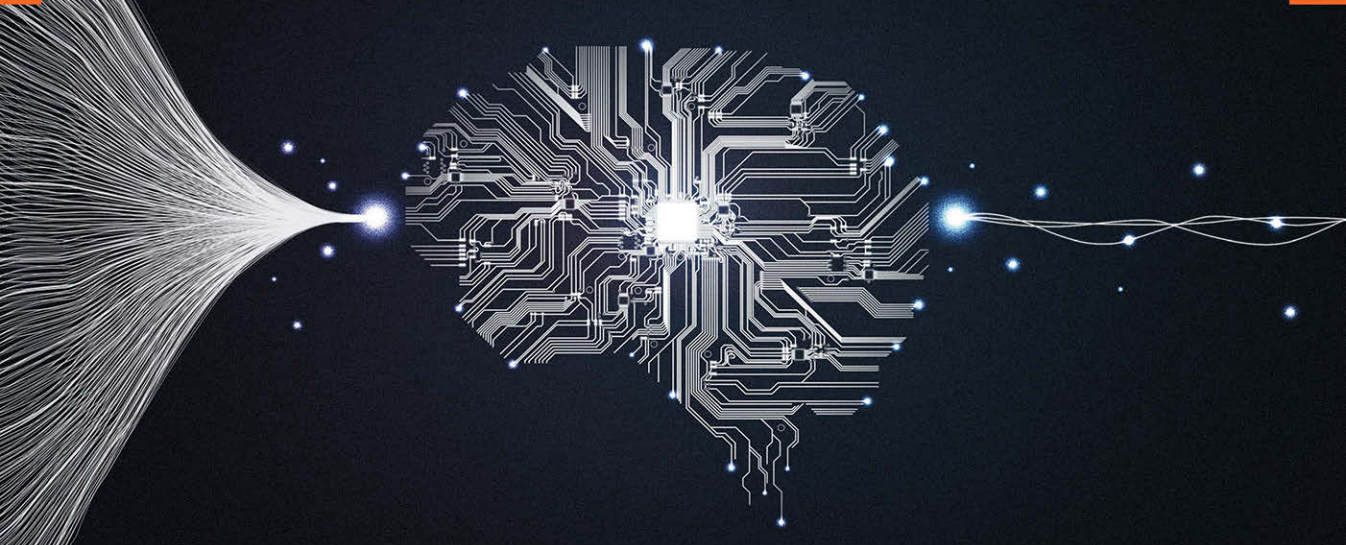
ADDISON WESLEY DATA & ANALYTICS SERIES

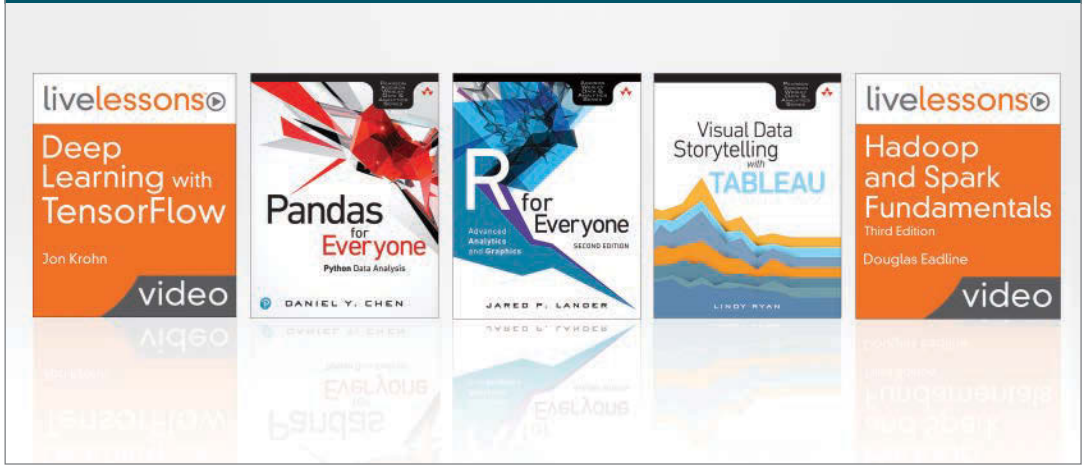
MACHINE LEARNING WITH PYTHON FOR EVERYONE



MARK E. FENNER

Machine Learning with Python for Everyone

The Pearson Addison-Wesley Data & Analytics Series



Visit **informit.com/awdataseries** for a complete list of available publications.

The **Pearson Addison-Wesley Data & Analytics Series** provides readers with practical knowledge for solving problems and answering questions with data. Titles in this series primarily focus on three areas:

1. **Infrastructure:** how to store, move, and manage data
2. **Algorithms:** how to mine intelligence or make predictions based on data
3. **Visualizations:** how to represent data and insights in a meaningful and compelling way

The series aims to tie all three of these areas together to help the reader build end-to-end systems for fighting spam; making recommendations; building personalization; detecting trends, patterns, or problems; and gaining insight from the data exhaust of systems and user interactions.

Machine Learning with Python for Everyone

Mark E. Fenner

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2019938761

Cover image: cono0430/Shutterstock

ISBN-13: 978-0-13-484562-3
ISBN-10: 0-13-484562-5

ScoutAutomatedPrintCode

To my son, Ethan—
with the eternal hope of a better tomorrow

This page intentionally left blank

Contents

This page intentionally left blank

Foreword

Whether it is called statistics, data science, machine learning, or artificial intelligence, learning patterns from data is transforming the world. Nearly every industry imaginable has been touched (or soon will be) by machine learning. The combined progress of both hardware and software improvements are driving rapid advancements in the field, though it is upon software that most people focus their attention.

While many languages are used for machine learning, including R, C/C++, Fortran, and Go, Python has proven remarkably popular. This is in large part thanks to scikit-learn, which makes it easy to not only train a host of different models but to also engineer features, evaluate the model quality, and score new data. The scikit-learn project has quickly become one of Python's most important and powerful software libraries.

While advanced mathematical concepts underpin machine learning, it is entirely possible to train complex models without a thorough background in calculus and matrix algebra. For many people, getting into machine learning through programming, rather than math, is a more attainable goal. That is precisely the goal of this book: to use Python as a hook into machine learning and then add in some math as needed. Following in the footsteps of *R for Everyone* and *Pandas for Everyone*, *Machine Learning with Python for Everyone* strives to be open and accessible to anyone looking to learn about this exciting area of math and computation.

Mark Fenner has spent years practicing the communication of science and machine learning concepts to people of varying backgrounds, honing his ability to break down complex ideas into simple components. That experience results in a form of storytelling that explains concepts while minimizing jargon and providing concrete examples. The book is easy to read, with many code samples so the reader can follow along on their computer.

With more people than ever eager to understand and implement machine learning, it is essential to have practical resources to guide them, both quickly and thoughtfully. Mark fills that need with this insightful and engaging text. *Machine Learning with Python for Everyone* lives up to its name, allowing people with all manner of previous training to quickly improve their machine learning knowledge and skills, greatly increasing access to this important field.

Jared Lander,
Series Editor

This page intentionally left blank

Preface

In 1983, the movie *WarGames* came out. I was a preteen and I was absolutely engrossed: by the possibility of a nuclear apocalypse, by the almost magical way the lead character interacted with computer systems, but mostly by the potential of machines that could *learn*. I spent years studying the strategic nuclear arsenals of the East and the West—fortunately with a naivete of a tweener—but it was almost ten years before I took my first serious steps in computer programming. Teaching a computer to do a set process was amazing. Learning the intricacies of complex systems and bending them around my curiosity was a great experience. Still, I had a large step forward to take. A few short years later, I worked with my first program that was explicitly designed to *learn*. I was blown away and I knew I found my intellectual home. I want to share the world of *computer programs that learn* with you.

Audience

Who do I think *you* are? I've written *Machine Learning with Python for Everyone* for the absolute beginner to machine learning. Even more so, you may well have very little college-level mathematics in your toolbox *and I'm not going to try to change that*. While many machine learning books are very heavy on mathematical concepts and equations, I've done my best to *minimize* the amount of mathematical luggage you'll have to carry. I do expect, given the book's title, that you'll have some basic proficiency in Python. If you can *read* Python, you'll be able to get a lot more out of our discussions. While many books on machine learning rely on mathematics, I'm relying on stories, pictures, and Python code to communicate with you. There *will* be the occasional equation. Largely, these can be skipped if you are so inclined. But, if I've done my job well, I'll have given you enough context around the equation to maybe—just *maybe*—understand what it is trying to say.

Why might you have this book in your hand? The least common denominator is that all of my readers want to *learn* about machine learning. Now, you might be coming from very different backgrounds: a student in an introductory computing class focused on machine learning, a mid-career business analyst who all of sudden has been thrust beyond the limits of spreadsheet analysis, a tech hobbyist looking to expand her interests, or a scientist needing to analyze data in a new way. Machine learning is permeating society. Depending on your background, *Machine Learning with Python for Everyone* has different things to offer you. Even a mathematically sophisticated reader who is looking to do a break-in to machine learning using Python can get a lot out of this book.

So, my goal is to take someone with an interest or need to do some machine learning and teach them the *process* and the most important *concepts* of machine learning in a concrete way using the Python scikit-learn library and some of its friends. You'll come

away with overall patterns, strategies, pitfalls, and gotchas that will be applicable in every learning system you ever study, build, or use.

Approach

Many books that try to explain mathematical topics, such as machine learning, do so by presenting equations as if they tell a story to the uninitiated. I think that leaves many of us—even those of us who like mathematics!—stuck. Personally, I build a far better mental picture of the process of machine learning by combining visual and verbal descriptions with *running code*. I'm a computer scientist at heart and by training. I love building things. Building things is how I know that I've reached a level where I *really* understand them. You might be familiar with the phrase, "If you really want to know something, teach it to someone." Well, there's a follow-on. "If you really want to know something, teach a computer to do it!" That's my take on how I'm going to teach you machine learning. With minimal mathematics, I want to give you the concepts behind the most important and frequently used machine learning tools and techniques. Then, I want you to immediately see how to make a computer do it. One note: we won't be programming these methods from scratch. We'll be standing on the shoulders of giants and using some very powerful, time-saving, prebuilt software libraries (more on that shortly).

We won't be covering all of these libraries in great detail—there is simply too much material to do that. Instead, we are going to be practical. We are going to use the best tool for the job. I'll explain enough to orient you in the concept we're using—and then we'll get to using it. For our mathematically inclined colleagues, I'll give pointers to more in-depth references they can pursue. I'll save most of this for end-of-the-chapter notes so the rest of us can skip it easily.

If you are flipping through this introduction, deciding if you want to invest time in this book, I want to give you some insight into things that are out-of-scope for us. We aren't going to dive into mathematical proofs or rely on mathematics to explain things. There are many books out there that follow that path and I'll give pointers to my favorites at the ends of the chapters. Likewise, I'm going to assume that you are fluent in basic- to intermediate-level Python programming. However, for more advanced Python topics—and things that show up from third-party packages like NumPy or Pandas—I'll explain enough of what's going on so that you can understand each technique and its context.

Overview

In **Part I**, we establish a foundation. I'll give you some verbal and conceptual introductions to machine learning in Chapter 1. In Chapter 2 we introduce and take a slightly different approach to some mathematical and computational topics that show up repeatedly in machine learning. Chapters 3 and 4 walk you through your first steps in building, training, and evaluating learning systems that classify examples (classifiers) and quantify examples (regressors).

Part II shifts our focus to the most important aspect of applied machine learning systems: evaluating the success of our system in a realistic way. Chapter 5 talks about general

evaluation techniques that will apply to all of our learning systems. Chapters 6 and 7 take those general techniques and add evaluation capabilities for classifiers and regressors.

Part III broadens our toolbox of learning techniques and fills out the components of a practical learning system. Chapters 8 and 9 give us additional classification and regression techniques. Chapter 10 describes *feature engineering*: how we smooth the edges of rough data into forms that we can use for learning. Chapter 11 shows how to chain multiple steps together as a single learner and how to tune a learner's inner workings for better performance.

Part IV takes us beyond the basics and discusses more recent techniques that are driving machine learning forward. We look at learners that are made up of multiple little learners in Chapter 12. Chapter 13 discusses learning techniques that incorporate automated feature engineering. Chapter 14 is a wonderful capstone because it takes the techniques we describe throughout the book and applies them to two particularly interesting types of data: images and text. Chapter 15 both reviews many of the techniques we discuss and shows how they relate to more advanced learning architectures—neural networks and graphical models.

Our main focus is on the techniques of machine learning. We will investigate a number of learning algorithms and other processing methods along the way. However, completeness is not our goal. We'll discuss the most common techniques and only glance briefly at the two large subareas of machine learning: graphical models and neural, or deep, networks. However, we will see how the techniques we focus on relate to these more advanced methods.

Another topic we won't cover is implementing specific learning algorithms. We'll build on top of the algorithms that are already available in scikit-learn and friends; we'll create larger solutions using them as components. Still, someone has to implement the gears and cogs inside the black-box we funnel data into. If you are really interested in implementation aspects, you are in good company: I love them! Have all your friends buy a copy of this book, so I can argue I need to write a follow-up that dives into these lower-level details.

Acknowledgments

I must take a few moments to thank several people that have contributed greatly to this book. My editor at Pearson, Debra Williams Cauley, has been instrumental in every phase of this book's development. From our initial meetings, to her probing for a topic that might meet both our needs, to gently shepherding me through many (many!) early drafts, to constantly giving me just enough of a push to keep going, and finally climbing the steepest parts of the mountain at its peak . . . through all of these phases, Debra has shown the highest degrees of professionalism. I can only respond with a heartfelt *thank you*.

My wife, Dr. Barbara Fenner, also deserves more praise and thanks than I can give her in this short space. In addition to the burdens that any partner of an author must bear, she *also* served as my primary draft reader *and* our intrepid illustrator. She did the hard work of drafting all of the non-computer-generated diagrams in this book. While this is not our first joint academic project, it has been turned into the longest. Her patience is, by all appearances, never ending. Barbara, *I thank you!*

My primary technical reader was Marilyn Roth. Marilyn was unfailingly positive towards even my most egregious errors. *Machine Learning with Python for Everyone* is immeasurably better for her input. *Thank you*.

I would also like to thank several members of Pearson's editorial staff: Alina Kirsanova and Dmitry Kirsanov, Julie Nahil, and many other behind-the-scenes folks that I didn't have the pleasure of meeting. This book would not exist without you and your hardworking professionalism. *Thank you*.

Publisher's Note

The text contains unavoidable references to color in figures. To assist readers of the print edition, color PDFs of figures are available for download at http://informit.com/title/9780134845623.

For formatting purposes, decimal values in many tables have been manually rounded to two place values. In several instances, Python code and comments have been slightly modified—all such modifications should result in valid programs.

Online resources for this book are available at https://github.com/mfenner1.

About the Author

Mark Fenner, PhD, has been teaching computing and mathematics to adult audiences—from first-year college students to grizzled veterans of industry—since 1999. In that time, he has also done research in machine learning, bioinformatics, and computer security. His projects have addressed design, implementation, and performance of machine learning and numerical algorithms; security analysis of software repositories; learning systems for user anomaly detection; probabilistic modeling of protein function; and analysis and visualization of ecological and microscopy data. He has a deep love of computing and mathematics, history, and adventure sports. When he is not actively engaged in writing, teaching, or coding, he can be found launching himself, with abandon, through the woods on his mountain bike or sipping a post-ride beer at a swimming hole. Mark holds a *nidan* rank in judo and is a certified Wilderness First Responder. He and his wife are graduates of Allegheny College and the University of Pittsburgh. Mark holds a PhD in computer science. He lives in northeastern Pennsylvania with his family and works through his company, Fenner Training and Consulting, LLC.

This page intentionally left blank

Part I

First Steps

This page intentionally left blank

1

Let's Discuss Learning

1.1 Welcome

From time to time, people trot out a tired claim that computers can "only do what they are told to do." The claim is taken to mean that computers can only do what their programmers know how to do *and* can explain to the computer. This claim is *false*. Computers can perform tasks that their programmers cannot explain to them. Computers can solve tasks that their programmers do not understand. We will break down this paradox with an example of a computer program that *learns*.

I'll start by discussing one of the oldest—if not the oldest known—examples of a programmed machine-learning system. I've turned this into a story, but it is rooted in historical facts. Arthur Samuel was working for IBM in the 1950s and he had an interesting problem. He had to test the big computing machines that were coming off the assembly line to make sure transistors didn't blow up when you turned a machine on and ran a program—people don't like smoke in their workplace. Now, Samuel quickly got bored with running simple toy programs and, like many computing enthusiasts, he turned his attention towards *games*. He built a computer program that let him play checkers against himself. That was fun for a while: he tested IBM's computers by playing checkers. But, as is often the case, he got bored playing two-person games solo. His mind began to consider the possibility of getting a good game of checkers against a *computer opponent*. Problem was, he wasn't good enough at checkers to explain good checkers strategies to a computer!

Samuel came up with the idea of having the computer *learn* how to play checkers. He set up scenarios where the computer could make moves and evaluate the costs and benefits of those moves. At first, the computer was bad, very bad. But eventually, the program started making progress. It was slow going. Suddenly, Samuel had a great two-for-one idea: he decided to let one computer play another and take himself out of the loop. Because the computers could make moves much faster than Samuel could enter his moves—let alone think about them—the result was many more cycles of "make a move and evaluate the outcome" per minute and hour and day.

Here is the amazing part. It didn't take very long for the computer opponent to be able to consistently beat Samuel. *The computer became a better checkers player than its programmer!* How on earth could this happen, if "computers can only do what they are told to do"? The answer to this riddle comes when we analyze *what the computer was told to*

do. What Samuel told the computer to do was not the *play-checkers* task; it was the *learn-to-play-checkers* task. Yes, we just went all *meta* on you. *Meta* is what happens when you take a picture of someone taking a picture (of someone else). Meta is what happens when a sentence refers to itself; the next sentence is an example. *This sentence has five words*. When we access the meta level, we step outside the box we were playing in and we get an entirely new perspective on the world. *Learning to play checkers*—a task that develops skill at another task—is a meta task. It lets us move beyond a limiting interpretation of the statement, *computers can only do what they are told*. Computers do what they are told, but they can be told to *develop a capability*. Computers can be told to learn.

1.2 Scope, Terminology, Prediction, and Data

There are many kinds of computational learning systems out there. The academic field that studies these systems is called *machine learning*. Our journey will focus on the current *wunderkind* of learning systems that has risen to great prominence: *learning from examples*. Even more specifically, we will mostly be concerned with *supervised learning from examples*. What is that? Here's an example. I start by giving you several photos of two animals you've never seen before—with apologies to Dr. Seuss, they might be a Lorax or a Who—and then I tell you which animal is in which photo. If I give you a new, unseen photo you might be able to tell me the type of animal in it. Congratulations, *you're doing great*! You just performed supervised learning from examples. When a computer is coaxed to learn from examples, the examples are presented a certain way. Each example is measured on a common group of attributes and we record the values for each attribute on each example. Huh?

Imagine—or glance at Figure 1.1—a cartoon character running around with a basket of different measuring sticks which, when held up to an object, return some characteristic of that object, such as *this vehicle has four wheels*, *this person has brown hair*, *the temperature of that tea is 180° F*, and so on *ad nauseam* (that's an archaic way of saying *until you're sick of my examples*).

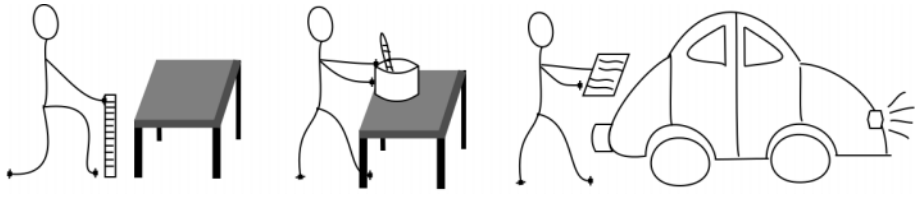


Figure 1.1 Humans have an insatiable desire to measure all sorts of things.

1.2.1 Features

Let's get a bit more concrete. For example—a meta-example, if you will—a dataset focused on human medical records might record several relevant values for each patient, such as height, weight, sex, age, smoking history, systolic and diastolic (that's the high and low numbers) blood pressures, and resting heart rate. The different people represented in the dataset are our examples. The biometric and demographic characteristics are our attributes.

We can capture this data very conveniently as in Table 1.1.

Table 1.1 A simple biomedical data table. Each row is an example. Each column contains values for a given attribute. Together, each attribute-value pair is a feature of an example.

patient id	height	weight	sex	age	smoker	hr	sys bp	dia bp
007	5'2"	120	M	11	no	75	120	80
2139	5'4"	140	F	41	no	65	115	75
1111	5'11"	185	M	41	no	52	125	75

Notice that each example—each row—is measured on the same attributes shown in the header row. The values of each attribute run down the respective columns.

We call the rows of the table the *examples* of the dataset and we refer to the columns as the *features*. Features are the measurements or values of our attributes. Often, people use "features" and "attributes" as synonyms describing the same thing; what they are referring to are the column of values. Still, some people like to distinguish among three concepts: *what-is-measured*, *what-the-value-is*, and *what-the-measured-value-is*. For those strict folks, the first is an attribute, the second is a value, and the last is a feature—an attribute and a value paired together. Again, we'll mostly follow the typical conversational usage and call the columns *features*. If we are specifically talking about *what-is-measured*, we'll stick with the term *attribute*. You will inevitably see both, used both ways, when you read about machine learning.

Let's take a moment and look at the types of values our attributes—what is measured—can take. One type of value distinguishes between different groups of people. We might see such groups in a census or an epidemiological medical study—for example, sex {*male*, *female*} or a broad record of ethnic-cultural-genetic heritage {*African*, *Asian*, *European*, *Native American*, *Polynesian*}. Attributes like these are called discrete, symbolic, categorical, or nominal attributes, but we are *not* going to stress about those names. If you struggled with those in a social science class, you are free to give a hearty huzzah.

Here are two important, or at least practical, points about categorical data. One point is that these values are discrete. They take a small, limited number of possibilities that typically represent one of several options. You're right that small and several are relative terms—just go with it. The second point is that the information in those attributes can be recorded in two distinct ways:

- As a single feature that takes one value for each option, *or*
- As several features, one per option, where one, and only one, of those features is marked as *yes* or *true* and the remainder are marked as *no* or *false*.

Here's an example. Consider

Name	Sex
Mark	Male
Barb	Female
Ethan	Male

versus:

Name	Sex is Female	Sex is Male
Mark	No	Yes
Barb	Yes	No
Ethan	No	Yes

If we had a column for community type in a census, the values might be Urban, Rural, and Suburban with three possible values. If we had the expanded, multicolumn form, it would take up three columns. Generally, we aren't motivated or worried about table size here. What matters is that some learning methods are, shall we say, particular in preferring one form or the other. There are other details to point out, but we'll save them for later.

Some feature values can be recorded and operated on as numbers. We may lump them together under the term *numerical* features. In other contexts, they are known as *continuous* or, depending on other details, *interval* or *ratio* values. Values for attributes like height and weight are typically recorded as decimal numbers. Values for attributes like age and blood pressure are often recorded as whole numbers. Values like counts—say, how many wheels are on a vehicle—are strictly whole numbers. Conveniently, we can perform arithmetic ($+, -, \times, /$) on these. While we *can* record categorical data as numbers, we can't necessarily perform *meaningful* numerical calculations directly on those values. If two states—say, Pennsylvania and Vermont—are coded as 2 and 14, it probably makes no sense to perform arithmetic on those values. There is an exception: if, by design, those values *mean* something beyond a unique identifier, we might be able to do some or all of the maths. For extra credit, you can find some meaning in the state values I used where mathematics would make sense.

1.2.2 Target Values and Predictions

Let's shift our focus back to the list of biomedical attributes we gathered. As a reminder, the column headings were height, weight, sex, age, smoker, heart rate, systolic blood pressure, and diastolic blood pressure. These attributes might be useful data for a health care provider trying to assess the likelihood of a patient developing cardiovascular heart. To do so, we would need another piece of information: did these folks develop heart disease?

If we have that information, we can add it to the list of attributes. We could capture and record the idea of "developing heart disease" in several different ways. Did the patient:

- Develop any heart disease within ten years: yes/no
- Develop X-level severity heart disease within ten years: None or Grade I, II, III
- Show some level of a specific indicator for heart disease within ten years: percent of coronary artery blockage

We could tinker with these questions based on resources at our disposal, medically relevant knowledge, and the medical or scientific puzzles we want to solve. Time is a precious resource; we might not have ten years to wait for an outcome. There might be medical knowledge about what percent of blockage is a critical amount. We could modify the time horizon or come up with different attributes to record.

In any case, we can pick a concrete, measurable target and ask, "Can we find a predictive relationship between the attributes we have *today* and the outcome that we will see *at some future time*?" We are literally trying to predict the future—maybe ten years from now—from things we know today. We call the concrete outcome our *target feature* or simply our *target*. If our target is a category like $\{sick, healthy\}$ or $\{None, I, II, III\}$, we call the process of learning the relationship *classification*. Here, we are using the term *classification* in the sense of finding the different classes, or categories, of a possible outcome. If the target is a smooth sweeping of numerical values, like the usual decimal numbers from elementary school $\{27.2, 42.0, 3.14159, -117.6\}$, we call the process *regression*. If you want to know why, go and google *Galton regression* for the history lesson.

We now have some handy terminology in our toolbox: most importantly *features*, both either *categorical* or *numerical*, and a *target*. If we want to emphasize the features being used to predict the future unknown outcome, we may call them *input features* or *predictive features*. There are a few issues I've swept under the carpet. In particular, we'll address some alternative terminology at the end of the chapter.

1.3 Putting the Machine in Machine Learning

I want you to create a mental image of a factory machine. If you need help, glance at Figure 1.2. On the left-hand side, there is a conveyor belt that feeds inputs into the machine. On the right-hand side, the machine spits out outputs which are words or numbers. The words might be *cat* or *dog*. The numbers might be $\{0, 1\}$ or $\{-2.1, 3.7\}$. The machine itself is a big hulking box of metal. We can't really see what happens on the inside. But we can see a control panel on the side of the machine, with an operator's seat in front of it. The control panel has some knobs we can set to numerical values and some switches we can flip on and off. By adjusting the knobs and switches, we can make different products appear on the right-hand side of the machine, depending on what came in the left-hand side. Lastly, there is a small side tray beside the operator's chair. The tray can be used to feed additional information, that is not easily captured by knobs and switches, into the machine. Two quick notes for the skeptical reader: our knobs *can* get us

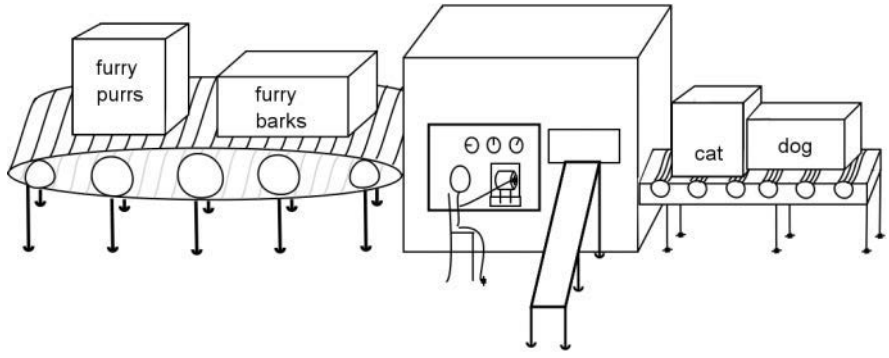


Figure 1.2 Descriptions go in. Categories or other values come out. We can adjust the machine to improve the relationship between the inputs and outputs.

arbitrarily small and large values ($-\infty$ to ∞, if you insist) and we don't *strictly* need on/off switches, since knobs set to precisely 0 or 1 could serve a similar purpose.

Moving forward, this factory image is a great entry point to understand how *learning algorithms* figure out relationships between features and a target. We can sit down as the machine operator and press a magic—probably green—*go* button. Materials roll in the machine from the left and *something* pops out on the right. Our curiosity gets the best of us and we twiddle the dials and flip the switches. Then, *different* things pop out the right-hand side. We turn up KnobOne and the machine pays more attention to the sounds that the input object makes. We turn down KnobTwo and the machine pays less attention to the number of limbs on the input object. If we have a goal—if there is some *known* product we'd like to see the machine produce—hopefully our knob twiddling gets us closer to it.

Learning algorithms are formal rules for how we manipulate our controls. After seeing examples where the target is known, learning algorithms take a given big-black-box and use a well-defined method to set the dials and switches to *good* values. While *good* can be quite a debatable quality in an ethics class, here we have a gold standard: our known target values. If they don't match, we have a problem. The algorithm adjusts the control panel settings so our *predicted outs* match the *known outs*. Our name for the machine is a *learning model* or just a *model*.

An example goes into the machine and, based on the settings of the knobs and switches, a class or a numerical value pops out. Do you want a different output value from the same input ingredients? Turn the knobs to different settings or flip a switch. One machine has a *fixed* set of knobs and switches. The knobs can be turned, but we can't add new knobs. If we add a knob, we have a *different machine*. Amazingly, the differences between knob-based learning methods boil down to answering three questions:

1. What knobs and switches are there: what is on the control panel?
2. How do the knobs and switches interact with an input example: what are the inner workings of the machine?

3. How do we set the knobs from some *known* data: how do we align the inputs with the outputs we want to see?

Many learning models that we will discuss can be described as machines with knobs and switches—with no need for the additional side input tray. Other methods require the side tray. We'll hold off discussing that more thoroughly, but if your curiosity is getting the best of you, flip to the discussion of *nearest neighbors* in Section 3.5.

Each learning method—which we imagine as a black-box factory machine and a way to set knobs on that machine—is really an *implementation* of an *algorithm*. For our purposes, an algorithm is a finite, well-defined sequence of steps to solve a task. An implementation of an algorithm is the specification of those steps in a particular programming language. The algorithm is the abstract idea; the implementation is the concrete existence of that idea—at least, as concrete as a computer program can be! In reality, algorithms *can* also be implemented in hardware—just like our factory machines; it's far easier for us to work with software.

1.4 Examples of Learning Systems

Under the umbrella of supervised learning from examples, there is a major distinction between two things: predicting values and predicting categories. Are we trying (1) to relate the inputs to one of a few possible categories indicated by discrete symbols, or (2) to relate the inputs to a more-or-less continuous range of numerical values? In short, is the target categorical or numerical? As I mentioned, predicting a category is called *classification*. Predicting a numerical value is called *regression*. Let's explore examples of each.

1.4.1 Predicting Categories: Examples of Classifiers

Classifiers are models that take input examples and produce an output that is one of a small number of possible groups or classes:

1. **Image Classification.** From an input image, output the animal (e.g., cat, dog, zebra) that is in the image, or none if there is no animal present. Image analysis of this sort is at the intersection of machine learning and computer vision. Here, our inputs will be a large collection of image files. They might be in different formats (png, jpeg, etc.). There may be substantial differences between the images: (1) they might be at different scales, (2) the animals may be centered or cut-off on the edges of the frame, and (3) the animals might be blocked by other things (e.g., a tree). These all represent challenges for a learning system—and for learning researchers! But, there are some nice aspects to image recognition. Our concept of *cat* and what images constitute a cat is fairly fixed. Yes, there could be blurred boundaries with animated cats—Hobbes, Garfield, Heathcliff, I'm looking at you—but short of evolutionary time scales, cat is a pretty static concept. We don't have a moving target: the *relationship* between the images and our concept of *cat* is fixed over time.

2. **Stock Action.** From a stock's price history, company fundamental data, and other relevant financial and market data, output whether we should buy or sell a stock. This problem adds some challenges. Financial records might only be available in text form. We might be interested in relevant news stories but we have to somehow figure out what's relevant—either by hand or (perhaps!) using another learning system. Once we've figured out the relevant text, we have to interpret it. These steps are where learning systems interact with the field of natural language processing (NLP). Continuing on with our larger task, we have a time series of data—repeated measurements over time. Our challenges are piling up. In financial markets, we probably have a moving target! What worked yesterday to pick a winning stock is almost certainly not going to work tomorrow in the exact same way. We may need some sort of method or technique that accounts for a changing relationship between the inputs and the output. Or, we may simply hope for the best and use a technique that assumes we don't have a moving target. Disclaimer: I am not a financial advisor nor am I offering investment advice.

3. **Medical Diagnosis.** From a patient's medical record, output whether they are sick or healthy. Here we have an even more complicated task. We might be dealing with a combination of text and images: medical records, notes, and medical imaging. Depending on context that may or may not be captured in the records—for example, traveling to tropical areas opens up the possibility of catching certain nasty diseases—different signs and symptoms may lead to very different diagnoses. Likewise, for all our vast knowledge of medicine, we are only beginning to understand some areas. It would be great for our system to read and study, like a doctor and researcher, the latest and greatest techniques for diagnosing patients. Learning to *learn-to-learn* is a meta-task in the extreme.

These are big-picture examples of classification systems. As of 2019, learning systems exist that handle many aspects of these tasks. We will even dive into basic image and language classifiers in Chapter 14. While each of these examples has its own domain-specific difficulties, they share a common task in building a model that separates the target categories in a useful and accurate way.

1.4.2 Predicting Values: Examples of Regressors

Numerical values surround us in modern life. Physical measurements (temperature, distance, mass), monetary values, percents, and scores are measured, recorded, and processed endlessly. Each can easily become a target feature that answers a question of interest:

1. **Student Success.** We could attempt to predict student scores on exams. Such a system might allow us to focus tutoring efforts on struggling students *before* an exam. We could include features like homework completion rates, class attendance, a measure of daily engagement or participation, and grades in previous courses. We could even include opened-ended written assessments and recommendations from prior instructors. As with many regression problems, we could reasonably convert

this regression problem to a classification problem by predicting a pass/fail or a letter grade instead of a raw numerical score.

2. **Stock Pricing.** Similar to the buy/sell stock classifier, we could attempt to predict the future price—dollar value—of a stock. This variation seems like a more difficult task. Instead of being satisfied with a broad estimate of *up* or *down*, we want to predict that the price will be $20.73 in two weeks. Regardless of difficulty, the inputs could be essentially the same: various bits of daily trading information and as much fundamental information—think quarterly reports to shareholders—as we'd like to incorporate.

3. **Web Browsing Behavior.** From an online user's browsing and purchasing history, predict (in percentage terms) how likely the user is to click on an advertising link or to purchase an item from an online store. While the input features of browsing and purchasing history are *not* numerical, our target—a percentage value—is. So, we have a regression problem. As in the image classification task, we have many small pieces of information that each contribute to the overall result. The pieces need context—how they relate to each other—to really become valuable.

1.5 Evaluating Learning Systems

Learning systems are rarely perfect. So, one of our key criteria is measuring how well they do. How *correct* are the predictions? Since nothing comes for free, we also care about the *cost* of making the predictions. What *computational resources* did we invest to get those predictions? We'll look at evaluating both of these aspects of learning system performance.

1.5.1 Correctness

Our key criteria for evaluating learning systems is that they give us correct predictive answers. If we didn't particularly care about correct answers, we could simply flip a coin, spin a roulette wheel, or use a random-number generator on a computer to get our output predictions. We want our learning system—that we are investing time and effort in building and running—to do better than random guessing. So, (1) we need to quantify how well the learner is doing and (2) we want to compare that level of success—or sadly, failure—with other systems. Comparing with other systems can even include comparisons with random guessers. There's a good reason to make that comparison: if we can't beat a random guesser, we need to go back to the drawing board—or maybe take a long vacation.

Assessing correctness is a surprisingly subtle topic which we will discuss in great detail throughout this book. But, for now, let's look at two classic examples of the difficulty of assessing correctness. In medicine, many diseases are—fortunately!—pretty rare. So, a doctor could get a large percentage of correct diagnoses by simply looking at every person in the street and saying, "that person doesn't have the rare disease." This scenario illustrates at least four issues that must be considered in assessing a potential diagnosis:

1. How common is an illness: what's the base rate of sickness?
2. What is the cost of *missing* a diagnosis: what if a patient isn't treated and gets gravely ill?
3. What is the cost of *making* a diagnosis? Further testing might be invasive and costly; worrying a patient needlessly could be very bad for a patient with high levels of anxiety.
4. Doctors typically diagnose patients that come into the office because they are symptomatic. That's a pretty significant difference from a random person in the street.

A second example comes from the American legal system, where there is a presumption of innocence and a relatively high bar for determining guilt. Sometimes this criteria is paraphrased as, "It is better for 99 criminals to go free than for 1 honest citizen to go to jail." As in medicine, we have the issue of rareness. Crime and criminals are relatively rare and getting rarer. We also have different costs associated with failures. We value clearing an honest citizen more than catching every criminal—at least that's how it works in the idealized world of a high-school civics class. Both these domains, legal and medical, deal with unbalanced target classes: disease and guilt are not 50–50 balanced outcomes. We'll talk about evaluating with unbalanced targets in Section 6.2.

One last issue in assessing correctness: two wrongs don't necessarily make a right. If we are predicting rainfall and, in one case, we underpredict by 2 inches while in another case we overpredict by 2 inches, these don't always cancel out. We cannot say, "On average, we were perfect!" Well, in fact, that's *strictly* true and it might be *good enough* in some instances. Usually, however, we do care, and very deeply in other cases, that we were wrong in both predictions. If we were trying to determine the amount of additional water to give some plants, we might end up giving plants a double dose of water that causes them to drown. Brown Thumbs—myself included—might like using that excuse in the next Great Garden Fiasco.

1.5.2 Resource Consumption

In our modern age of disposable everything, it is tempting to apply a consumer-driven strategy to our learning systems: if we hit a barrier, just buy our way through it. Data storage is extremely cheap. Access to phenomenally powerful hardware, such as a computing cluster driven by graphics processors, is just an email or an online purchase away. This strategy begs a question: shouldn't we just throw more hardware at problems that hit resource limits?

The answer *might* be yes—but let's, at least, use quantitative data to make that decision. At each level of increased complexity of a computational system, we pay for the privilege of using that more complex system. We need more software support. We need more specialized human capital. We need more complicated off-the-shelf libraries. We lose the ability to rapidly prototype ideas. For each of these costs, we need to justify the expense. Further, for many systems, there are small portions of code that are a performance bottleneck. It is often possible to maintain the simplicity of the overall system and only have a small kernel that draws on more sophisticated machinery to go *fast*.

With all that said, there are two primary resources that we will measure: time and memory. How long does a computation take? What is the maximum memory it needs? It is often the case that these can be traded off one for the other. For example, I can precompute the answers to common questions and, *presto*, I have very quick answers available. This, however, comes at the cost of writing down those answers and storing them somewhere. I've reduced the time needed for a computation but I've increased my storage requirements.

If you've ever used a lookup table—maybe to convert lengths from imperial to metric—you've made use of this tradeoff. You *could* pull out a calculator, plug values into a formula, and get an answer for *any* specific input. Alternatively, you can just flip through a couple pages of tables and find a precomputed answer up to some number of digits. Now, since the formula method here is quite fast to begin with, we actually end up losing out by using a big table. If the formula were more complicated and expensive to compute, the table could be a big time saver.

A physical-world equivalent of precomputation is when chefs and mixologists premake important components of larger recipes. Then, when they need lime juice, instead of having to locate limes, clean them, and juice them, they simply pull a lime juice cube out of the freezer or pour some lime juice out of a jar. They've traded time at the beginning and some storage space in the refrigerator for faster access to lime juice to make your killer mojito or guacamole.

Likewise, a common computation technique called compression trades off time for space. I can spend some time finding a smaller, compact representation of *Moby Dick*—including the dratted chapter on cetology (the study of whales)—and store the compressed text instead of the raw text of the tome. Now, my hard drive or my bookshelf is less burdened by storage demands. Then, when I get a craving to read about 19th-century whaling adventures, I can do so. But first, I have to pay a computation cost in time because I have to decompress the book. Again, there is a tradeoff between computational time and storage space.

Different learning systems make different tradeoffs between what they *remember* about the data and *how long* they spend processing the data. From one perspective, learning algorithms compress data in a way that is suitable for predicting new examples. Imagine that we are able to take a large data table and reduce it to a few knobs on a machine: as long as we have a copy of that machine around, we only need a few pieces of information to recreate the table.

1.6 A Process for Building Learning Systems

Even in this brief introduction to learning systems, you've probably noticed that there are many, many options that describe a learning system.

- There are different domains where we might apply learning, such as business, medicine, and science.

- There are different tasks within a domain, such as animal image recognition, medical diagnosis, web browsing behavior, and stock market prediction.
- There are different types of data.
- There are different models relating features to a target.

We haven't yet explicitly discussed the different types of models we'll use, but we will in the coming chapters. Rest assured, there are many options.

Can we capture any generalities about building learning systems? Yes. Let's take two different perspectives. First, we'll talk at a high level where we are more concerned with the world around the learning system and less concerned with the learning system itself. Second, we'll dive into some details at a lower level: imagine that we've abstracted away all the complexity of the world around us and are just trying to make a learning system go. More than that, we're trying to find a solid relationship between the features and the target. Here, we've reduced a very open-ended problem to a defined and constrained learning task.

Here are the high-level steps:

1. Develop an understanding of our task (task understanding).
2. Collect and understand our data (data collection).
3. Prepare the data for modeling (data preparation).
4. Build models of relationships in the data (modeling).
5. Evaluate and compare one or more models (evaluation).
6. Transition the model into a deployable system (deployment).

These steps are shown in Figure 1.3. I'll insert a few common caveats here. First, we normally have to iterate, or repeat, these steps. Second, some steps may feed back to prior steps. As with most real-world processes, progress isn't always a straight line forward. These steps are taken from the CRISP-DM flow chart that organizes the high-level steps of building a learning system. I've renamed the first step from *business understanding* to *task understanding* because not all learning problems arise in the business world.

Within the high-level modeling step—that's step 4 above—there are a number of important choices for a supervised learning system:

1. What part of the data is our target and what are the features?
2. What sort of machine, or learning model, do we want to use to relate our input features to our target feature?
3. Do the data and machine have any negative interactions? If so, do we need to do additional data preparation as part of our model building?
4. How do we set the knobs on the machine? What is our algorithm?

While these breakdowns can help us organize our thoughts and discussions about learning systems, they are not the final story. Let's inform the emperor and empress that they are missing their clothes. Abstract models or flow-chart diagrams can never capture the messy reality of the real world. In the real world, folks building learning systems are often called in (1) after there is already a pile of data gathered and (2) after some primary-stake holders—ahem, bosses—have already decided what they want done. From our humble perspective—and from what I want you to get out of this book—that's just

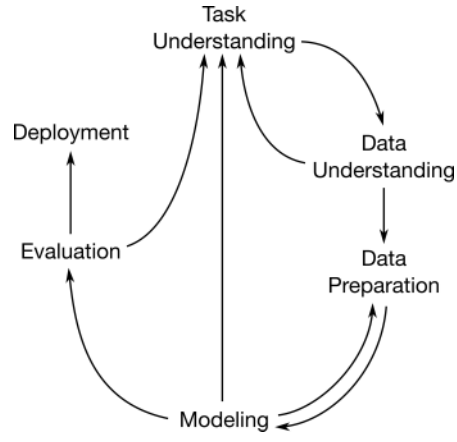


Figure 1.3 A high-level view of machine learning.

fine. We're not going to dive into the details of collecting data, experimental design, and determining good business, engineering, or scientific relationships to capture. We're just going to say, "Go!" We will move from that pile of data to usable examples, applying different learning systems, evaluating the results, and comparing alternatives.

1.7 Assumptions and Reality of Learning

Machine learning is not magic. I can see the look of shock on your faces. But seriously, learning cannot go beyond some fundamental limits. What are those limits? Two of them are directly related to the data we have available. If we are trying to predict heart disease, having information about preferred hair styles and sock colors is—likely—*not* going to help us build a useful model. If we have *no* useful features, we'll only pick up on illusory patterns—random noise—in the data. Even with useful features, in the face of many irrelevant features, learning methods may bog down and stop finding useful relationships. Thus, we have a fundamental limit: we need features that are relevant to the task at hand.

The second data limit relates to quantity. An entire subject called *computational learning theory* is devoted to the details of telling us how many examples we need to learn relationships under certain mathematically idealized conditions. From a practical standpoint, however, the short answer is *more*. We want *more* data. This rule-of-thumb is often summarized as *data is greater than (more important than) algorithms*: data > algorithms. There's truth there, but as is often the case, the details matter. If our data is excessively noisy— whether due to errors or randomness—it might not actually be useful. Bumping up to a stronger learning machine—sort of like bumping up a weight class in wrestling or getting a larger stone bowl for the kitchen—*might* get us better results. Yet, you can be bigger and not *necessarily* better: you might not be a more winning wrestler or make a better guacamole just because you are stronger or have better tools.

Speaking of errors in measurements, not every value we have in a data table is going to be 100% accurate. Our measuring rulers might be off by a bit; our ruler-readers might be rounding off their measurements in different ways. Worse yet, we might ask questions in surveys and receive *lies* in response—the horror! Such is reality. Even when we measure with great attention to detail, there can be differences when we repeat the process. Mistakes and uncertainty happen. The good news is that learning systems *can* tolerate these foibles. The bad news is that with enough *noise* it can be impossible to pick out intelligible patterns.

Another issue is that, in general, we don't know *every* relevant piece of information. Our outcomes may not be known with *complete* accuracy. Taken together, these give us unaccounted-for differences when we try to relate inputs to outputs. Even if we have *every* relevant piece of information measured with *perfect* accuracy, some processes in the world are *fundamentally* random—quarks, I'm looking at you. If the random-walk stock market folks are right, the pricing of stocks is random in a very deep sense. In more macro-scale phenomena, the randomness may be less fundamental but it still exists. If we are missing a critical measurement, it may appear as if the relationship in our data is random. This loss of perspective is like trying to live in a three-dimensional world while only seeing two-dimensional shadows. There are many 3D objects that can give the *same* 2D shadow when illuminated from various angles; a can, a ball, and a coffee cup are all circles from the bird's eye view (Figure 1.4). In the same way, missing measurements can obscure the real, detectable nature of a relationship.

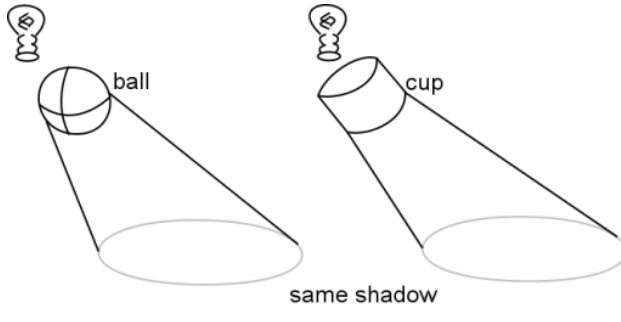


Figure 1.4 Perspective can shape our view of reality.

Now for two last technical caveats that we'll hold throughout this book. One is that the relationship between the features and the target is not, itself, a moving target. For example, over time the factors that go into a successful business have presumably changed. In industrial businesses, you need access to raw materials, so being in the right place and the right time is a massive competitive advantage. In knowledge-driven enterprises, the ability to attract high-quality employees from a relatively small pool of talent is a strong competitive advantage. Higher-end students of the mathematical arts call relationships that don't change over time *stationary learning tasks*. Over time, or at least over different examples in our dataset, the underlying relationship is assumed to—we act as if it does—remain the same.

The second caveat is that we don't necessarily assume that nature operates the same way as our machine. All we care about is matching the inputs and the outputs. A more scientific model may seek to *explain* the relationship between inputs and outputs with a mathematical formula that represents *physical laws* of the universe. We aren't going down that rabbit hole. We are content to capture a surface view—a black box or gift-wrapped present—of the relationship. We have our cake, but we can't eat it too.

1.8 End-of-Chapter Material

1.8.1 The Road Ahead

There isn't much to summarize in an introductory chapter. Instead, I'll talk a bit about what we're going through in the four parts of this book.

Part I will introduce you to several types of learning machines and the basics of evaluating and comparing them. We'll also take a brief look at some mathematical topics and concepts that you need to be familiar with to deal with our material. Hopefully, the math is presented in a way that doesn't leave you running for the exits. As you will see, I use a different approach and I hope it will work for you.

Part II dives into detail about evaluating learning systems. My belief is that the biggest risk in developing learning systems is lying to ourselves about how well we are doing. Incidentally, the second biggest risk is blindly using a system without respect for the evolving and complex systems that surround it. Specifically, components in a complex socio-technical system are not swappable like parts in a car. We also need to tread very carefully with the assumption that the future is like the past. As for the first issue, after I get you up and running with some practical examples, we'll take on the issue of evaluation immediately. As to the second issue—good luck with that! In all seriousness, it is beyond the scope of this book and it requires great experience and wisdom to deal with data that acts differently in different scenarios.

Part III fleshes out a few more learning methods and then shifts focus towards manipulating the data so we can use our various learning methods more effectively. We then turn our focus towards fine-tuning methods by manipulating their internal machinery—diving into their inner workings.

Part IV attacks some issues of increasing complexity: dealing with inadequate vocabulary in our data, using images or text instead of a nice table of examples and features, and making learners that are themselves composed of multiple sublearners. We finish by highlighting some connections between different learning systems and with some seemingly far more complicated methods.

1.8.2 Notes

If you want to know more about Arthur Samuel, this brief bio will get you started: http://history.computer.org/pioneers/samuel.html.

The idea of the *meta* level and self-reference is fundamental to higher computer science, mathematics, and philosophy. For a brilliant and broad-reaching look at *meta*, check out

Godel, Escher, Bach: An Eternal Golden Braid by Hofstadter. It is long, but intellectually rewarding.

There are many alternative terms for what we call features and target: inputs/outputs, independent/dependent variables, predictors/outcome, etc.

PA and VT were the 2nd and 14th states to join the United States.

What makes the word *cat* mean the object ⋆CAT⋆ and how is this related to the attributes that we take to define a cat: meowing, sleeping in sunbeams, etc.? To dive into this topic, take a look at Wittgenstein (https://plato.stanford.edu/entries/wittgenstein), particularly on language and meaning.

The examples I discussed introduce some of the *really hard* aspects of learning systems. In many cases, this book is about the *easy* stuff (running algorithms) plus some *medium*-difficulty components (feature engineering). The real world introduces complexities that are *hard*.

Outside of supervised learning from examples, there are several other types of learning. *Clustering* is not supervised learning although it does use examples. We will touch on it in later chapters. Clustering looks for patterns in data without specifying a special target feature. There are other, wilder varieties of learning systems; analytic learning and inductive logic programming, case-based reasoning, and reinforcement learning are some major players. See Tom Mitchell's excellent book titled *Machine Learning*. Incidentally, Mitchell has an excellent breakdown of the steps involved in constructing a learning system (the modeling step of the CRISP-DM process).

Speaking of CRISP-DM, Foster Provost and Tom Fawcett have a great book *Data Science for Business Understanding* that dives into machine learning and its role in organizations. Although their approach is focused on the business world, anyone who has to make use of a machine-learning system that is part of a larger system or organization—that's most of them, folks—can learn many valuable lessons from their book. They also have a great approach to tackling technical material. I highly recommend it.

There are many issues that make real-world data hard to process. Missing values is one of these issues. For example, data may be missing randomly, or it may be missing in concert with other values, or it may be missing because our data isn't really a good sample of all the data we might consider. Each of these may require different steps to try to fill in the missing values.

Folks that have background in the social sciences might be wondering why I didn't adopt the classical distinctions of nominal, ordinal, interval, and ratio data. The answer is twofold. First, that breakdown of types misses some important distinctions; search the web for "level of measurement" to get you started. Second, our use of modeling techniques will convert categories, whether ordered or not, to numerical values and then do their thing. Those types aren't treated in a fundamentally different way. However, there are statistical techniques, such as *ordinal regression*, that can account for ordering of categories.

2

Some Technical Background

2.1 About Our Setup

We're about to get down—funk style—with some coding. The chapters in this book started life as Jupyter notebooks. If you're unfamiliar with Jupyter notebooks, they are a very cool environment for working with Python code, text, and graphics in one browser tab. Many Python-related blogs are built with Jupyter notebooks. At the beginning of each chapter, I'll execute some lines of code to set up the coding environment.

The content of `mlwpy.py` is shown in Appendix A. While `from module import *` is generally not recommended, in this case I'm using it specifically to get all of the definitions in `mlwpy.py` included in our notebook environment without taking up forty lines of code. Since scikit-learn is *highly modularized*—which results in many, many `import` lines—the `import *` is a nice way around a long setup block in every chapter. `%matplotlib inline` tells the notebook system to display the graphics made by Python inline with the text.

```
In [1]:
```

```
from mlwpy import *
%matplotlib inline
```

2.2 The Need for Mathematical Language

It is very difficult to talk about machine learning (ML) without discussing *some* mathematics. Many ML textbooks take that to an extreme: they are *math* textbooks that happen to discuss machine learning. I'm going to flip that script on its head. I want you to *understand* the math we use and to have some intuition from daily life about what the math-symbol-speak means when you see it. I'm going to minimize the amount of math that I throw around. I also want us—that's you and me together on this wonderful ride—to see the math as code before, or very shortly after, seeing it as mathematical symbols.

Maybe, just maybe, after doing all that you *might* decide you want to dig into the mathematics more deeply. Great! There are endless options to do so. But that's not our

game. We care more about the ideas of machine learning than using high-end math to express them. Thankfully, we only need a few ideas from the mathematical world:

- Simplifying equations (algebra),
- A few concepts related to randomness and chance (probability),
- Graphing data on a grid (geometry), and
- A compact notation to express some arithmetic (symbols).

Throughout our discussion, we'll use some algebra to write down ideas precisely and without unnecessary verbalisms. The ideas of probability underlie many machine learning methods. Sometimes this is *very* direct, as in Naive Bayes (NB); sometimes it is less direct, as in Support Vector Machines (SVMs) and Decision Trees (DTs). Some methods rely very directly on a geometric description of data: SVMs and DTs shine here. Other methods, such as NB, require a bit of squinting to see how they can be viewed through a geometric lens. Our bits of notation are pretty low-key, but they amount to a specialized vocabulary that allows us to pack ideas into boxes that, in turn, fit into larger packages. If this sounds to you like refactoring a computer program from a single monolithic script into modular functions, give yourself a prize. That's *exactly* what is happening.

Make no mistake: a deep dive into the arcane mysteries of machine learning requires more, and deeper, mathematics than we will discuss. However, the ideas we *will* discuss are the first steps and the conceptual foundation of a more complicated presentation. Before taking those first steps, let's introduce the major Python packages we'll use to make these abstract mathematical ideas concrete.

2.3 Our Software for Tackling Machine Learning

The one tool I expect you to have in your toolbox is a basic understanding of good, old-fashioned procedural programming in Python. I'll do my best to discuss any topics that are more intermediate or advanced. We'll be using a few modules from the Python standard library that you may not have seen: `itertools`, `collections`, and `functools`.

We'll also be making use of several members of the Python number-crunching and data science stack: `numpy`, `pandas`, `matplotlib`, and `seaborn`. I won't have time to teach you all the details about these tools. However, we won't be using their more complicated features, so nothing should be too mind-blowing. We'll also briefly touch on one or two other packages, but they are relatively minor players.

Of course, much of the reason to use the number-crunching tools is because they form the foundation of, or work well with, scikit-learn. `sklearn` is a great environment for playing with the ideas of machine learning. It implements many different learning algorithms and evaluation strategies and gives you a uniform interface to run them. Win, win, and win. If you've never had the struggle—pleasure?—of integrating several different command-line learning programs . . . you didn't miss anything. Enjoy your world, it's a better place. A side note: scikit-learn is the project's name; `sklearn` is the name of the Python package. People use them interchangeably in conversation. I usually write `sklearn` because it is shorter.

2.4 Probability

Most of us are practically exposed to probability in our youth: rolling dice, flipping coins, and playing cards all give concrete examples of *random events*. When I roll a standard six-sided die—you role-playing gamers know about all the *other*-sided dice that are out there—there are six different outcomes that can happen. Each of those *events* has an equal chance of occurring. We say that the probability of each event is $\frac{1}{6}$. Mathematically, if I—a Roman numeral one, not me, myself, and I—is the case where we roll a one, we'll write that as $P(I) = \frac{1}{6}$. We read this as "the probability of rolling a one is one-sixth."

We can roll dice in Python in a few different ways. Using NumPy, we can generate evenly weighted random events with `np.random.randint`. `randint` is designed to mimic Python's indexing semantics, which means that we *include* the starting point and we *exclude* the ending point. The practical upshot is that if we want values from 1 to 6, we need to start at 1 and end at 7: the 7 will not be included. If you are more mathematically inclined, you can remember this as a half-open interval.

In [2]:

```
np.random.randint(1, 7)
```

Out[2]:

```
4
```

If we want to convince ourselves that the numbers are really being generated with equal likelihoods (as with a perfect, fair die), we can draw a chart of the frequency of the outcomes of many rolls. We'll do that in three steps. We'll roll a die, either a few times or many times:

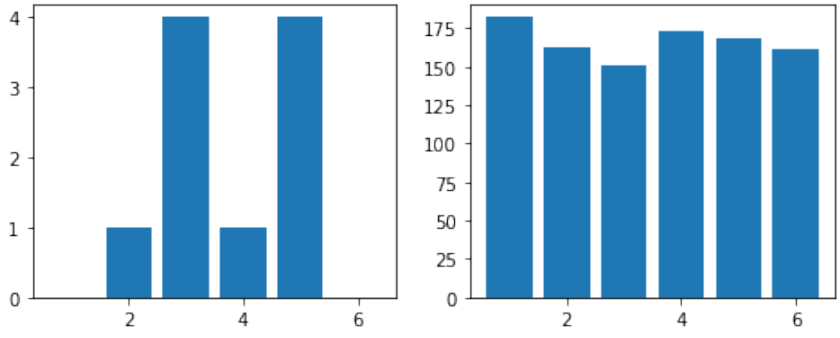
In [3]:

```
few_rolls  = np.random.randint(1, 7, size=10)
many_rolls = np.random.randint(1, 7, size=1000)
```

We'll count up how many times each event occurred with `np.histogram`. Note that `np.histogram` is designed around plotting buckets of continuous values. Since we want to capture discrete values, we have to create a bucket that surrounds our values of interest. We capture the ones, *I*, by making a bucket between 0.5 and 1.5.

In [4]:

```
few_counts  = np.histogram(few_rolls,  bins=np.arange(.5, 7.5))[0]
many_counts = np.histogram(many_rolls, bins=np.arange(.5, 7.5))[0]

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 3))
ax1.bar(np.arange(1, 7), few_counts)
ax2.bar(np.arange(1, 7), many_counts);
```



There's an important lesson here. When dealing with random events and overall behavior, a small sample can be misleading. We may need to crank up the number of examples—rolls, in this case—to get a better picture of the underlying behavior. You might ask why I didn't use `matplotlib`'s built-in `hist` function to make the graphs in one step. `hist` works well enough for larger datasets that take a wider range of values but, unfortunately, it ends up obfuscating the simple case of a few discrete values. Try it out for yourself.

2.4.1 Primitive Events

Before experimenting, we assumed that the probability of rolling a one is one out of six. That number comes from $\frac{\text{\#ways this event can occur}}{\text{\#of different events}}$. We can test our understanding of that ratio by asking, "What is the probability of rolling an odd number?" Well, using Roman numerals to indicate the outcomes of a roll, the odd numbers in our space of events are *I*, *III*, *V*. There are three of these and there are six total primitive events. So, we have $P(\text{odd}) = \frac{3}{6} = \frac{1}{2}$. Fortunately, that gels with our intuition.

We can approach this calculation a different way: an odd can occur in three ways and those three ways don't overlap. So, we can add up the individual event probabilities: $P(\text{odd}) = P(I) + P(III) + P(V) = \frac{1}{6} + \frac{1}{6} + \frac{1}{6} = \frac{3}{6} = \frac{1}{2}$. We can get probabilities of *compound events* by *either* counting primitive events *or* adding up the probabilities of primitive events. It's the same thing done in two different ways.

This basic scenario gives us an *in* to talk about a number of important aspects of probability:

- The sum of the probabilities of all possible primitive events in a universe is 1. $P(I) + P(II) + P(III) + P(IV) + P(V) + P(VI) = 1$.
- The probability of an event *not* occurring is 1 minus the probability of it occurring. $P(\text{even}) = 1 - P(\text{not even}) = 1 - P(\text{odd})$. When discussing probabilities, we often write "not" as ¬, as in $P(\neg\text{even})$. So, $P(\neg\text{even}) = 1 - P(\text{even})$.
- There are nonprimitive events. Such a compound event is a combination of primitive events. The event we called *odd* joined together three primitive events.

- A roll will be even or odd, but not both, and all rolls are either even or odd. These two compound events cover all the possible primitive events without any overlap. So, $P(\text{even}) + P(\text{odd}) = 1$.

Compound events are also recursive. We can create a compound event from other compound events. Suppose I ask, "What is the probability of getting an odd or a value greater than 3 or *both*?" That group of events, taken together, is a larger group of primitive events. If I attack this by counting those primitive events, I see that the odds are odd $= \{I, III, V\}$ and the big values are big $= \{IV, V, VI\}$. Putting them together, I get $\{I, III, IV, V, VI\}$ or $\frac{5}{6}$. The probability of this compound event is a bit different from the probability of *odds* being $\frac{1}{2}$ and the probability of *greater-than-3* being $\frac{1}{2}$. I can't just add those probabilities. Why not? Because I'd get a sum of one—meaning we covered everything—but that only demonstrates the error. The *reason* is that the two compound events overlap: they share primitive events. Rolling a five, *V*, occurs in both subgroups. Since they overlap, we can't just add the two together. We have to add up everything in both groups individually and then remove one of whatever was double- counted. The double-counted events were in both groups—they were odd *and* big. In this case, there is just one double-counted event, *V*. So, removing them looks like $P(\text{odd}) + P(\text{big}) - P(\text{odd and big})$. That's $\frac{1}{2} + \frac{1}{2} - \frac{1}{6} = \frac{5}{6}$.

2.4.2 Independence

If we roll two dice, a few interesting things happen. The two dice don't communicate or act on one another in any way. Rolling a *I* on one die does not make it more or less likely to roll any value on the other die. This concept is called independence: the two events—rolls of the individual dice—are independent of each other.

For example, consider a different set of outcomes where each event is the *sum* of the rolls of two dice. Our sums are going to be values between 2 (we roll two *I*s) and 12 (we roll two *VI*s). What is the probability of getting a sum of 2? We can go back to the counting method: there are 36 total possibilities (6 for each die, times 2) and the only way we can roll a total of 2 is by rolling two *I*s which can *only* happen one way. So, $P(2) = \frac{1}{36}$. We can also reach that conclusion—because the dice don't communicate or influence each other—by rolling *I* on die 1 and *I* on die 2, giving $P(I_1)P(I_2) = \frac{1}{6} \cdot \frac{1}{6} = \frac{1}{36}$. If events are independent, we can multiply their probabilities to get the joint probability of both occurring. *Also*, if we multiply the probabilities and we get the same probability as the overall resulting probability we calculated by counting, we *know* the events must be independent. Independent probabilities work both ways: they are an *if-and-only-if*.

We can combine the ideas of (1) summing the probabilities of different events and (2) the independence of events, to calculate the probability of getting a total of three $P(3)$. Using the event counting method, we figure that this event can happen in two different ways: we roll (I, II) or we roll (II, I) giving $2/36 = 1/18$. Using probability calculations, we can write:

$$\begin{aligned} P(3) &= P((I, II)) + P((II, I)) \\ &= P(I)P(II) + P(II)P(I) \\ &= \frac{1}{6} \cdot \frac{1}{6} + \frac{1}{6} \cdot \frac{1}{6} = \frac{2}{36} = \frac{1}{18} \end{aligned}$$

Phew, that was a lot of work to verify the answer. Often, we can make use of shortcuts to reduce the number of calculations we have to perform. Sometimes these shortcuts are from knowledge of the problem and sometimes they are clever applications of the rules of probability we've seen so far. If we see multiplication, we can mentally think about the two-dice scenario. If we have a scenario like the dice, we can multiply.

2.4.3 Conditional Probability

Let's create one more scenario. In classic probability-story fashion, we will talk about two *urns*. Why urns? I guess that, before we had buckets, people had urns. So, if you don't like urns, you can think about buckets. I digress.

The first urn U_I has three red balls and one blue ball in it. The second urn U_{II} has two red balls and two blue balls. We flip a coin and then we pull a ball out of an urn. If the coin comes up heads, we pick from U_I; otherwise, we pick from U_{II}. We end up at U_I half the time and then we pick a red ball $\frac{3}{4}$ of those times. We end up at U_{II} the other half of the time and we pick a red ball $\frac{2}{4}$ of those times. This scenario is like wandering down a path with a few intersections. As we walk along, we are presented with a different set of options at the next crossroads.

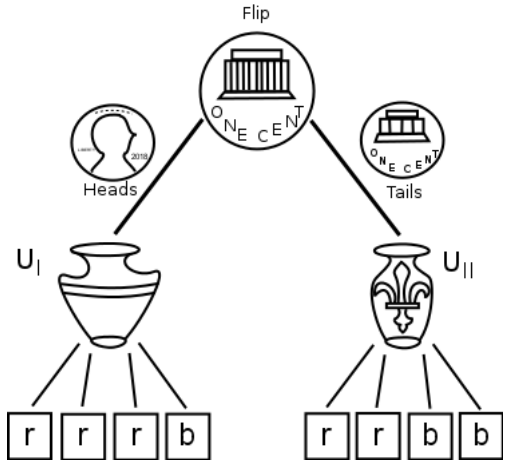


Figure 2.1 A two-step game from coins to urns.

If we sketch out the paths, it looks like Figure 2.1. If we count up the possibilities, we will see that under the whole game, we have five red outcomes and three blue outcomes. $P(\text{red}) = \frac{5}{8}$. Simple, right? Not so fast, speedy! This counting argument *only* works when we have equally likely choices at each step. Imagine we have a very wacky coin that causes me to end up at Urn *I* 999 out of 1000 times: then our chances of picking a red ball would end up quite close to the chance of just picking a red ball from Urn *I*. It would be similar to *almost* ignoring the existence of Urn *II*. We should account for this difference and, at the same time, make use of updated information that might come along the way.

If we play a partial game and we know that we're at Urn I—for example, after we've flipped a head in the first step—our odds of picking a red ball are different. Namely, the probability of picking a red ball—*given* that we are picking from Urn I—is $\frac{3}{4}$. In mathematics, we write this as $P(\text{red} \mid U_I) = \frac{3}{4}$. The vertical bar, $\mid$, is read as "given". Conditioning—a commonly verbed noun in machine learning and statistics—constrains us to a subset of the primitive events that could possibly occur. In this case, we condition on the occurrence of a head on our coin flip.

How often do we end up picking a red ball from Urn I? Well, to do that we have to (1) get to Urn I by flipping a head, and then (2) pick a red ball. Since the coin doesn't affect the events in Urn I—it picked Urn I, not the balls *within* Urn I—the two are independent and we can multiply them to find the joint probability of the two events occurring. So, $P(\text{red and } U_I) = P(\text{red} \mid U_I)P(U_I) = \frac{1}{2} \cdot \frac{3}{4} = \frac{3}{8}$. The order here may seem a bit weird. I've written it with the later event—the event that depends on U_I—first and the event that kicks things off, U_I, second. This order is what you'll usually see in written mathematics. Why? Largely because it places the $\mid U_I$ next to the $P(U_I)$. You can think about it as reading from the bottom of the diagram back towards the top.

Since there are two nonoverlapping ways to pick a red ball (either from Urn I or from Urn II), we can *add* up the different possibilities. Just as we did for Urn I, for Urn II we have $P(\text{red and } U_{II}) = P(\text{red} \mid U_{II})P(U_{II}) = \frac{1}{2} \cdot \frac{2}{4} = \frac{2}{8}$. Adding up the alternative ways of getting red balls, either out of Urn I or out of Urn II, gives us: $P(\text{red}) = P(\text{red} \mid U_I)P(U_I) + P(\text{red} \mid U_{II})P(U_{II}) = \frac{3}{8} + \frac{2}{8} = \frac{5}{8}$. *Mon dieu*! At least we got the same answer as we got by the *simple* counting method. But now, you know what that important vertical bar, $P(\ \mid\ )$, means.

2.4.4 Distributions

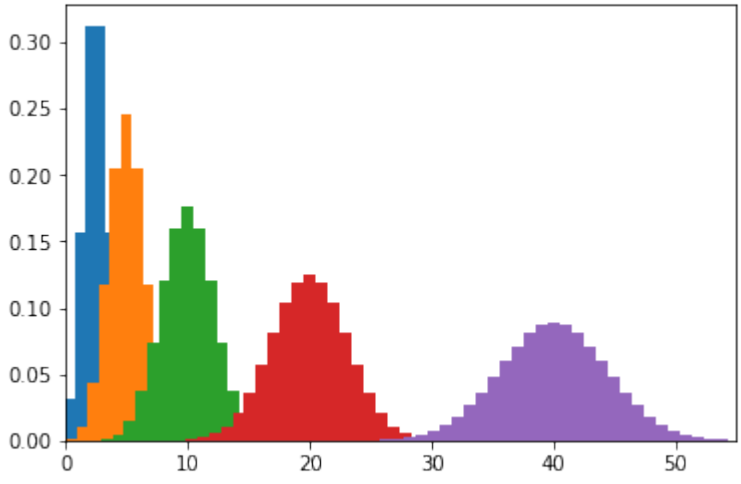
There are many different ways of assigning probabilities to events. Some of them are based on direct, real-world experiences like dice and cards. Others are based on hypothetical scenarios. We call the mapping between events and probabilities a *probability distribution*. If you give me an event, then I can look it up in the probability distribution and tell you the probability that it occurred. Based on the rules of probability we just discussed, we can also calculate the probabilities of more complicated events. When a group of events shares a common probability value—like the different faces on a fair die—we call it a *uniform distribution*. Like Storm Troopers in uniform, they all look the same.

There is one other, very common distribution that we'll talk about. It's so fundamental that there are multiple ways to approach it. We're going to go back to coin flipping. If I flip a coin many, many times and count the number of heads, here's what happens as we increase the number of flips:

In [5]:

```
import scipy.stats as ss

b = ss.distributions.binom
for flips in [5, 10, 20, 40, 80]:
```

```
    # binomial with .5 is result of many coin flips
    success = np.arange(flips)
    our_distribution = b.pmf(success, flips, .5)
    plt.hist(success, flips, weights=our_distribution)
plt.xlim(0, 55);
```

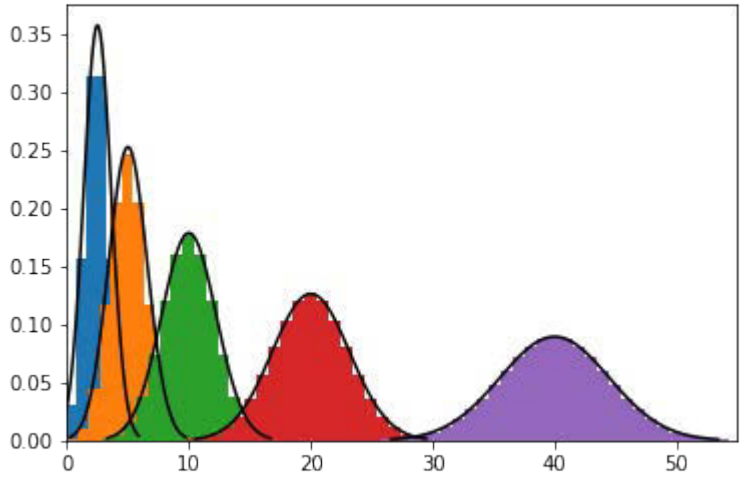


If I ignore that the whole numbers are *counts* and replace the graph with a smoother curve that takes values everywhere, instead of the stair steps that climb or descend at whole numbers, I get something like this:

In [6]:

```
b = ss.distributions.binom
n = ss.distributions.norm

for flips in [5, 10, 20, 40, 80]:
    # binomial coin flips
    success = np.arange(flips)
    our_distribution = b.pmf(success, flips, .5)
    plt.hist(success, flips, weights=our_distribution)

    # normal approximation to that binomial
    # we have to set the mean and standard deviation
    mu      = flips * .5,
    std_dev = np.sqrt(flips * .5 * (1-.5))

    # we have to set up both the x and y points for the normal
```

```
    # we get the ys from the distribution (a function)
    # we have to feed it xs, we set those up here
    norm_x = np.linspace(mu-3*std_dev, mu+3*std_dev, 100)
    norm_y = n.pdf(norm_x, mu, std_dev)
    plt.plot(norm_x, norm_y, 'k');

plt.xlim(0, 55);
```



You can think about increasing the number of coin flips as increasing the accuracy of a measurement—we get more decimals of accuracy. We see the difference between 4 and 5 out of 10 and then the difference between 16, 17, 18, 19, and 20 out of 40. Instead of a big step, it becomes a smaller, more gradual step. The step-like sequences become progressively better approximated by the smooth curves. Often, these smooth curves are called *bell-shaped curves*—and, to keep the statisticians happy, yes, there are other bell-shaped curves out there. The specific bell-shaped curve that we are stepping towards is called the *normal distribution*.

The *normal distribution* has three important characteristics:

1. Its midpoint has the most likely value—the hump in the middle.
2. It is symmetric—can be mirrored—about its midpoint.
3. As we get further from the midpoint, the values fall off more and more quickly.

There are a variety of ways to make these characteristics mathematically precise. It turns out that with suitable mathese and small-print details, those characteristics also lead to the normal distribution—the smooth curve we were working towards! My mathematical colleagues may cringe, but the primary feature we need from the normal distribution is its shape.

2.5 Linear Combinations, Weighted Sums, and Dot Products

When mathematical folks talk about a linear combination, they are using a technical term for what we do when we check out from the grocery store. If your grocery store bill looks like:

Product	Quantity	Cost Per
Wine	2	12.50
Orange	12	.50
Muffin	3	1.75

you can figure out the total cost with some arithmetic:

In [7]:

```
(2 * 12.50) + (12 * .5) + (3 * 1.75)
```

Out[7]:

```
36.25
```

We might think of this as a *weighted sum*. A *sum* by itself is simply adding things up. The total number of items we bought is:

In [8]:

```
2 + 12 + 3
```

Out[8]:

```
17
```

However, when we buy things, we pay for each item based on its cost. To get a total cost, we have to add up a sequence of costs times quantities. I can phrase that in a slightly different way: we have to weight the quantities of different items by their respective prices. For example, each orange costs $0.50 and our total cost for oranges is $6. Why? Besides the invisible hand of economics, the grocery store does not want us to pay the same amount of money for the bottle of wine as we do for an orange! In fact, we don't want that either: $10 oranges aren't really a thing, are they? Here's a concrete example:

In [9]:

```
# pure python, old-school
quantity = [2, 12, 3]
costs    = [12.5, .5, 1.75]
partial_cost = []
for q,c in zip(quantity, costs):
```

```
    partial_cost.append(q*c)
sum(partial_cost)
```

Out[9]:

```
36.25
```

In [10]:

```
# pure python, for the new-school, cool kids
quantity = [2, 12, 3]
costs    = [12.5, .5, 1.75]
sum(q*c for q,c in zip(quantity,costs))
```

Out[10]:

```
36.25
```

Let's return to computing the total cost. If I line up the quantities and costs in NumPy arrays, I can run the same calculation. I can also get the benefits of data that is more organized under the hood, concise code that is easily extendible for more quantities and costs, and better small- and large-scale performance. *Whoa!* Let's do it.

In [11]:

```
quantity = np.array([2, 12, 3])
costs    = np.array([12.5, .5, 1.75])
np.sum(quantity * costs) # element-wise multiplication
```

Out[11]:

```
36.25
```

This calculation can also be performed by NumPy with `np.dot`. `dot` multiplies the elements pairwise, selecting the pairs in lockstep down the two arrays, and then adds them up:

In [12]:

```
print(quantity.dot(costs),       # dot-product way 1
      np.dot(quantity, costs),   # dot-product way 2
      quantity @ costs,          # dot-product way 3
                                 # (new addition to the family!)
      sep='\n')
```

```
36.25
36.25
36.25
```

If you were ever exposed to dot products and got completely lost when your teacher started discussing geometry and cosines and vector lengths, I'm so, so sorry! Your teacher wasn't wrong, but the idea is no more complicated than checking out of the grocery store. There are two things that make the linear combination (expressed in a dot product): (1) we multiply the values pairwise, and (2) we add up all those subresults. These correspond to (1) a single multiplication to create subtotals for each line on a receipt and (2) adding those subtotals together to get your final bill.

You'll also see the dot product written mathematically (using q for `quantity` and c for `cost`) as $\sum_i q_i c_i$. If you haven't seen this notation before, here's a breakdown:

1. The $\sum$, a capital Greek *sigma*, means add up,
2. The $q_i c_i$ means multiply two things, and
3. The i ties the pieces together in lockstep like a sequence index.

More briefly, it says, "add up *all* of the element-wise multiplied q and c." Even more briefly, we might call this the *sum product* of the quantities and costs. At our level, we can use sum product as a synonym for dot product.

So, combining NumPy on the left-hand side and mathematics on the right-hand side, we have:

$$\texttt{np.dot(quantity,cost)} = \sum_i q_i c_i$$

Sometimes, that will be written as briefly as qc. If I want to emphasize the dot product, or remind you of it, I'll use a bullet ($\bullet$) as its symbol: $q \bullet c$. If you are uncertain about the element-wise or lockstep part, you can use Python's `zip` function to help you out. It is designed precisely to march, in lockstep, through multiple sequences.

```
In [13]:
```

```
for q_i, c_i in zip(quantity, costs):
    print("{:2d} {:5.2f} --> {:5.2f}".format(q_i, c_i, q_i * c_i))

print("Total:",
      sum(q*c for q,c in zip(quantity,costs))) # cool-kid method
```

```
 2 12.50 --> 25.00
12  0.50 -->  6.00
 3  1.75 -->  5.25
Total: 36.25
```

Remember, we normally let NumPy—via `np.dot`—do that work for us!

2.5.1 Weighted Average

You might be familiar with a simple average—and now you're wondering, "What is a weighted average?" To help you out, the simple average—also called the mean—is an

equally weighted average computed from a set of values. For example, if I have three values $(10, 20, 30)$, I divide up my weights equally among the three values and, *presto*, I get thirds: $\frac{1}{3}10 + \frac{1}{3}20 + \frac{1}{3}30$. You might be looking at me with a distinct side eye, but if I rearrange that as $\frac{10+20+30}{3}$ you might be happier. I simply do `sum(values)/3`: add them all up and divide by the number of values. Look what happens, however, if I go back to the more expanded method:

In [14]:

```
values  = np.array([10.0, 20.0, 30.0])
weights = np.full_like(values, 1/3) # repeated (1/3)

print("weights:", weights)
print("via mean:", np.mean(values))
print("via weights and dot:", np.dot(weights, values))
```

```
weights: [0.3333 0.3333 0.3333]
via mean: 20.0
via weights and dot: 20.0
```

We can write the mean as a weighted sum—a sum product between values and weights. If we start playing around with the weights, we end up with the concept of *weighted averages*. With weighted averages, instead of using equal portions, we break the portions up any way we choose. In some scenarios, we insist that the portions add up to one. Let's say we weighted our three values by $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{4}$. Why might we do this? These weights could express the idea that the first option is *valued twice as much* as the other two and that the other two are valued equally. It might also mean that the first one is *twice as likely* in a random scenario. These two interpretations are close to what we would get if we applied those weights to underlying costs or quantities. You can view them as two sides of the same double-sided coin.

In [15]:

```
values  = np.array([10,  20,  30])
weights = np.array([.5, .25, .25])

np.dot(weights, values)
```

Out[15]:

```
17.5
```

One special weighted average occurs when the values are the different outcomes of a random scenario and the weights represent the probabilities of those outcomes. In this case, the weighted average is called the *expected value* of that set of outcomes. Here's a simple game. Suppose I roll a standard six-sided die and I get $1.00 if the die turns out odd and I lose $.50 if the die comes up even. Let's compute a dot product between the payoffs and the probabilities of each payoff. My expected outcome is to make:

In [16]:

```
                  # odd, even
payoffs = np.array([1.0, -.5])
probs   = np.array([ .5,  .5])
np.dot(payoffs, probs)
```

Out[16]:

```
0.25
```

Mathematically, we write the expected value of the game as $\mathrm{E}(\text{game}) = \sum_i p_i v_i$ with p being the probabilities of the events and v being the values or payoffs of those events. Now, in any *single* run of that game, I'll *either* make \$1.00 or lose \$.50. But, if I were to play the game, say 100 times, I'd expect to come out ahead by about \$25.00—the expected gain per game times the number of games. In reality, this outcome is a *random* event. Sometimes, I'll do better. Sometimes, I'll do worse. But \$25.00 is my best guess before heading into a game with 100 tosses. With many, many tosses, we're *highly likely* to get *very close* to that expected value.

Here's a simulation of 10000 rounds of the game. You can compare the outcome with `np.dot(payoffs, probs) * 10000`.

In [17]:

```
def is_even(n):
    # if remainder 0, value is even
    return n % 2 == 0

winnings = 0.0
for toss_ct in range(10000):
    die_toss = np.random.randint(1, 7)
    winnings += 1.0 if is_even(die_toss) else -0.5
print(winnings)
```

```
2542.0
```

2.5.2 Sums of Squares

One other, very special, sum-of-products is when both the *quantity* and the *value* are *two copies of the same thing*. For example, $5 \cdot 5 + (-3) \cdot (-3) + 2 \cdot 2 + 1 \cdot 1 = 5^2 + 3^2 + 2^2 + 1^2 = 25 + 9 + 4 + 1 = 39$. This is called a *sum of squares* since each element, multiplied by itself, gives the square of the original value. Here is how we can do that in code:

In [18]:

```
values = np.array([5, -3, 2, 1])
squares = values * values # element-wise multiplication
print(squares,
      np.sum(squares),   # sum of squares.  ha!
      np.dot(values, values), sep="\n")
```

```
[25  9  4  1]
39
39
```

If I wrote this mathematically, it would look like: `dot(values,values)` $= \sum_i v_i v_i = \sum_i v_i^2$.

2.5.3 Sum of Squared Errors

There is another very common summation pattern, the sum of squared errors, that fits in here nicely. In this case of mathematical terminology, the red herring is both *red* and a *herring*. If I have a known value `actual` and I have your guess as to its value `predicted`, I can compute your error with `error = predicted - actual`.

Now, that error is going to be positive or negative based on whether you over- or underestimated the actual value. There are a few mathematical tricks we can pull to make the errors positive. They are useful because when we measure errors, we don't want two wrongs—overestimating by 5 and underestimating by 5—to cancel out and make a right! The trick we will use here is to square the error: an error of $5 \rightarrow 25$ and an error of $-5 \rightarrow 25$. If you ask about your total squared error after you've guessed 5 and -5, it will be $25 + 25 = 50$.

In [19]:

```
errors = np.array([5, -5, 3.2, -1.1])
display(pd.DataFrame({'errors':errors,
                      'squared':errors*errors}))
```

	errors	squared
0	5.0000	25.0000
1	-5.0000	25.0000
2	3.2000	10.2400
3	-1.1000	1.2100

So, a squared error is calculated by $\text{error}^2 = (\text{predicted} - \text{actual})^2$. And we can add these up with $\sum_i (\text{predicted}_i - \text{actual}_i)^2 = \sum_i \text{error}_i^2$. This sum reads left to right as, "the sum of (open paren) errors which are squared (close paren)." It can be said more succinctly: the sum of squared errors. That looks a lot like the `dot` we used above:

In [20]:

```
np.dot(errors, errors)
```

Out[20]:

```
61.45
```

Weighted averages and sums of squared errors are probably the most common summation forms in machine learning. By knowing these two forms, you are now prepared to understand what's going on mathematically in many different learning scenarios. In fact, much of the notation that obfuscates machine learning from beginners—while that same notation *facilitates* communication amongst experts!—is really just compressing these summation ideas into fewer and fewer symbols. You now know how to pull those ideas apart.

You might have a small spidey sense tingling at the back of your head. It might be because of something like this: $c^2 = a^2 + b^2$. I can rename or rearrange those symbols and get $\text{distance}^2 = \text{len}_1^2 + \text{len}_2^2$ or $\text{distance} = \sqrt{\text{run}^2 + \text{rise}^2} = \sqrt{\sum_i x_i^2}$. Yes, our old friends—or nemeses, if you prefer—Euclid and Pythagoras can be wrapped up as a sum of squares. Usually, the a and b are distances, and we can compute distances by subtracting two values—just like we do when we compare our actual and predicted values. Hold on to your seats. An error is just a length—a distance—between an actual and a predicted value!

2.6 A Geometric View: Points in Space

We went from checking out at the grocery store to discussing sums of squared errors. That's quite a trip. I want to start from another simple daily scenario to discuss some basic geometry. I promise you that this will be the least geometry-class-like discussion of geometry you have ever seen.

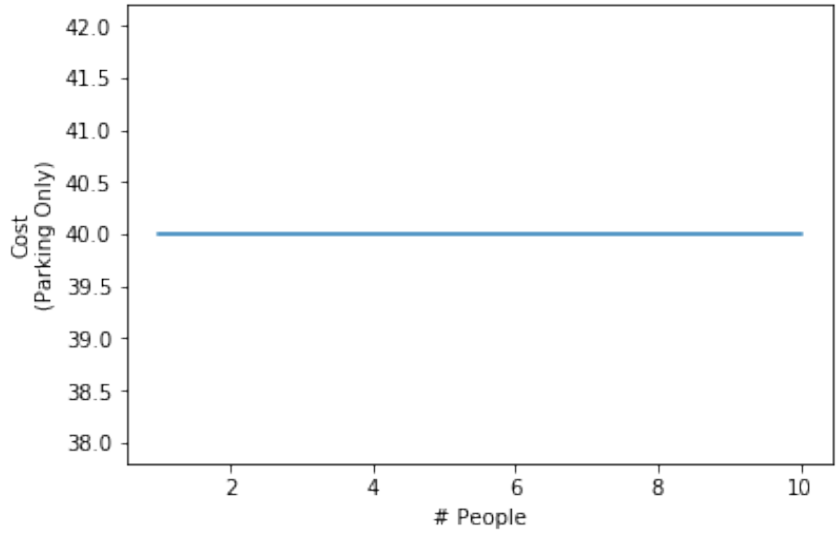
2.6.1 Lines

Let's talk about the cost of going to a concert. I hope that's suitably nonacademic. To start with, if you drive a car to the concert, you have to park it. For up to 10 (good) friends going to the show, they can all fit in a minivan—packed in like a clown car, if need be. The group is going to pay one flat fee for parking. That's good, because the cost of parking is usually pretty high: we'll say $40. Let's put that into code and pictures:

In [21]:

```
people = np.arange(1, 11)
total_cost = np.ones_like(people) * 40.0

ax = plt.gca()

ax.plot(people, total_cost)
```

```
ax.set_xlabel("# People")
ax.set_ylabel("Cost\n(Parking Only)");
```



In a math class, we would write this as total_cost $= 40.0$. That is, regardless of the number of people—moving back and forth along the x-axis at the bottom—we pay the same amount. When mathematicians start getting abstract, they reduce the expression to simply $y = 40$. They will talk about this as being "of the form" $y = c$. That is, the height or the y-value is equal to *some constant*. In this case, it's the value 40 everywhere. Now, it doesn't do us much good to park at the show and not buy tickets—although there is something to be said for tailgating. So, what happens if we have to pay \$80 per ticket?

In [22]:

```
people = np.arange(1, 11)
total_cost = 80.0 * people + 40.0
```

Graphing this is a bit more complicated, so let's make a table of the values first:

In [23]:

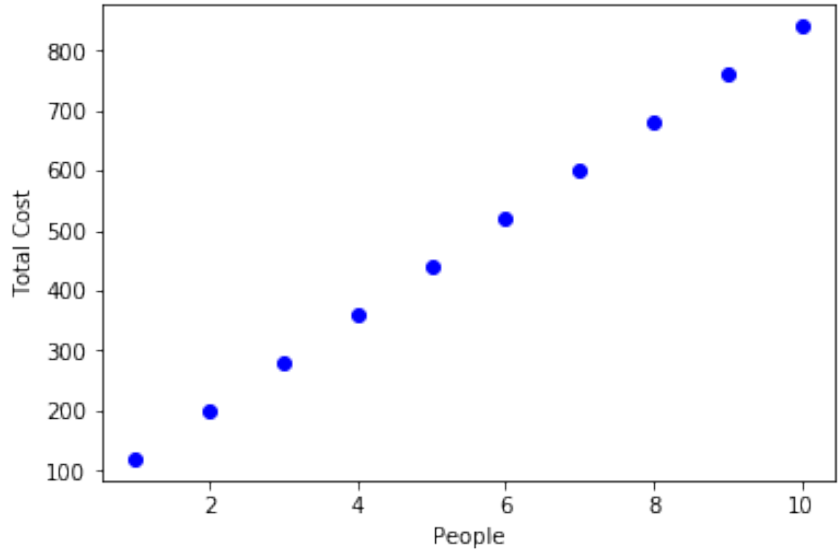
```
# .T (transpose) to save vertical space in printout
display(pd.DataFrame({'total_cost':total_cost.astype(np.int)},
                     index=people).T)
```

	1	2	3	4	5	6	7	8	9	10
total_cost	120	200	280	360	440	520	600	680	760	840

And we can plot that, point-by-point:

In [24]:

```
ax = plt.gca()
ax.plot(people, total_cost, 'bo')
ax.set_ylabel("Total Cost")
ax.set_xlabel("People");
```



So, if we were to write this in a math class, it would look like:

$$\text{total\_cost} = \text{ticket\_cost} \times \text{people} + \text{parking\_cost}$$

Let's compare these two forms—a constant and a line—and the various ways they might be written in Table 2.1.

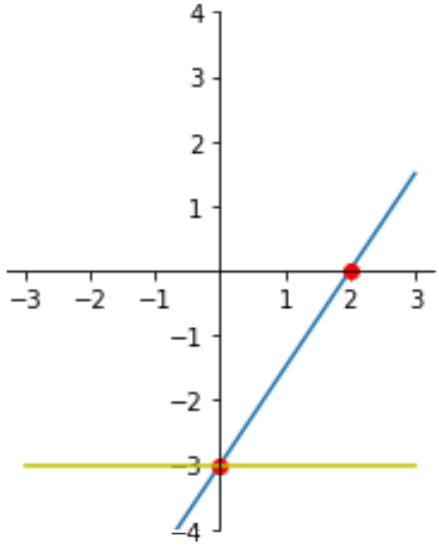
Table 2.1 Examples of constants and lines at different levels of language.

Name	Example	Concrete	Abstract	Mathese
Constant	total = parking	total = \$40	$y = 40$	$y = c$
Line	total = ticket × person + parking	total = 80 × person + 40	$y = 80x + 40$	$y = mx + b$

I want to show off one more plot that emphasizes the two defining components of the lines: m and b. The m value—which was the \$80 ticket price above—tells how much more we pay for each person we add to our trip. In math-speak, it is the *rise*, or increase in y for a single-unit increase in x. A unit increase means that the number of people on the x-axis goes from x to $x + 1$. Here, I'll control m and b and graph it.

In [25]:

```
# paint by number
# create 100 x values from -3 to 3
xs = np.linspace(-3, 3, 100)

# slope (m) and intercept (b)
m, b = 1.5, -3

ax = plt.gca()

ys = m*xs + b
ax.plot(xs, ys)

ax.set_ylim(-4, 4)
high_school_style(ax) # helper from mlwpy.py

ax.plot(0, -3,'ro') # y-intercept
ax.plot(2,  0,'ro') # two steps right gives three steps up

# y = mx + b with m=0 gives y = b
ys = 0*xs + b
ax.plot(xs, ys, 'y');
```



Since our slope is 1.5, taking two steps to the *right* results in us gaining three steps *up*. Also, if we have a *line* and we set the slope of the line m to 0, all of a sudden we are back

to a constant. Constants are a specific, restricted type of *horizontal* line. Our yellow line which passes through $y = -3$ is one.

We can combine our ideas about `np.dot` with our ideas about lines and write some slightly different code to draw this graph. Instead of using the pair (m, b), we can write an array of values $w = (w_1, w_0)$. One trick here: I put the w_0 second, to line up with the b. Usually, that's how it is written in mathese: the w_0 is the constant.

With the ws, we can use `np.dot` *if* we augment our `xs` with an extra column of ones. I'll write that augmented version of `xs` as `xs_p1` which you can read as "exs plus a column of ones." The column of ones serves the role of the 1 in $y = mx + b$. Wait, you don't see a 1 there? Let me rewrite it: $y = mx + b = mx + b \cdot 1$. See how I rewrote $b \to b \cdot 1$? That's the same thing we need to do to make `np.dot` happy. `dot` wants to multiply *something* times w_1 and *something* times w_0. We make sure that whatever gets multiplied by w_0 is a 1.

I call this process of tacking on a column of ones the *plus-one trick* or *+1 trick* and I'll have more to say about it shortly. Here's what the plus-one trick does to our raw data:

In [26]:

```
# np.c_[] lets us create an array column-by-column
xs    = np.linspace(-3, 3, 100)
xs_p1 = np.c_[xs, np.ones_like(xs)]

# view the first few rows
display(pd.DataFrame(xs_p1).head())
```

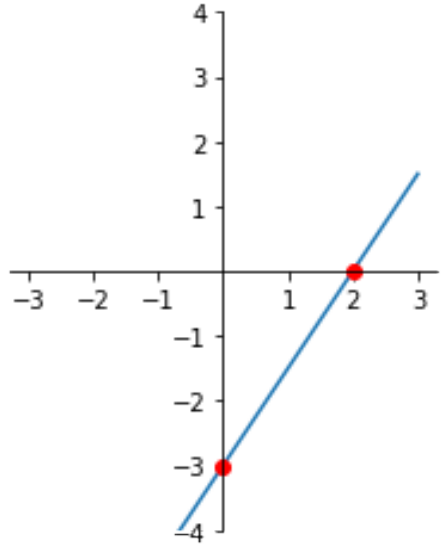
	0	1
0	-3.0000	1.0000
1	-2.9394	1.0000
2	-2.8788	1.0000
3	-2.8182	1.0000
4	-2.7576	1.0000

Now, we can combine our data and our weights very concisely:

In [27]:

```
w  = np.array([1.5, -3])
ys = np.dot(xs_p1, w)

ax = plt.gca()
ax.plot(xs, ys)

# styling
ax.set_ylim(-4, 4)
high_school_style(ax)
```

```
ax.plot(0, -3,'ro')  # y-intercept
ax.plot(2,  0,'ro'); # two steps to the right should be three whole steps up
```



Here are the two forms we used in the code: `ys = m*xs + b` and `ys = np.dot(xs_p1, w)`. Mathematically, these look like $y = mx + b$ and $y = wx^+$. Here, I'm using x^+ as an abbreviation for the x that has ones tacked on to it. The two forms defining `ys` mean the same thing. They just have some differences when we implement them. The first form has each of the components standing on its own. The second form requires x^+ to be augmented with a 1 and allows us to conveniently use the dot product.

2.6.2 Beyond Lines

We can extend the idea of lines in at least two ways. We can progress to wiggly curves and polynomials—equations like $f(x) = x^3 + x^2 + x + 1$. Here, we have a *more complex* computation on *one input value* x. Or, we can go down the road to multiple dimensions: planes, hyperplanes, and beyond! For example, in $f(x, y, z) = x + y + z$ we have *multiple input values* that we combine together. Since we will be *very* interested in multivariate data—that's multiple inputs—I'm going to jump right into that.

Let's revisit the rock concert scenario. What happens if we have more than one kind of item we want to purchase? For example, you might be surprised to learn that people like to consume beverages at concerts. Often, they like to consume what my mother affectionately refers to as "root beer." So, what if we have a cost for parking, a cost for the tickets, and a cost for each root beer our group orders. To account for this, we need a new formula. With `rb` standing for root beer, we have:

$$\text{total\_cost} = \text{ticket\_cost} \times \text{number\_people} + \text{rb\_cost} \times \text{number\_rbs} + \text{parking\_cost}$$

If we plug in some known values for parking cost, cost per ticket, and cost per root beer, then we have something more concrete:

$$\text{total\_cost} = 80 \times \text{number\_people} + 10 \times \text{number\_rbs} + 40$$

With one item, we have a simple two-dimensional plot of a line where one axis direction comes from the input "how many people" and the other comes from the output "total cost". With two items, we now have two *how many's* but still only one `total_cost`, for a total of three dimensions. Fortunately, we can still draw that somewhat reasonably. First, we create some data:

In [28]:

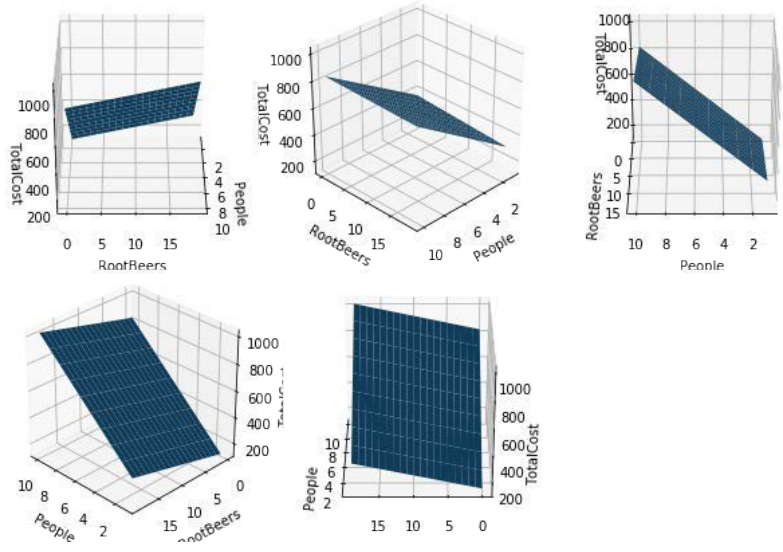
```
number_people = np.arange(1, 11) # 1-10 people
number_rbs    = np.arange(0, 20) # 0-19 rootbeers

# numpy tool to get cross-product of values (each against each)
# in two paired arrays.  try it out: np.meshgrid([0, 1], [10, 20])
# "perfect" for functions of multiple variables
number_people, number_rbs = np.meshgrid(number_people, number_rbs)

total_cost = 80 * number_people + 10 * number_rbs + 40
```

We can look at that data from a few different angles—literally. Below, we show the same graph from five different viewpoints. Notice that they are all flat surfaces, but the apparent tilt or slope of the surface looks different from different perspectives. The flat surface is called a plane.

In [29]:

```
# import needed for 'projection':'3d'
from mpl_toolkits.mplot3d import Axes3D
fig,axes = plt.subplots(2, 3,
                        subplot_kw={'projection':'3d'},
                        figsize=(9, 6))

angles = [0, 45, 90, 135, 180]
for ax,angle in zip(axes.flat, angles):
    ax.plot_surface(number_people, number_rbs, total_cost)
    ax.set_xlabel("People")
    ax.set_ylabel("RootBeers")
    ax.set_zlabel("TotalCost")
    ax.azim = angle

# we don't use the last axis
```

```
axes.flat[-1].axis('off')
fig.tight_layout()
```



It is pretty straightforward, in code and in mathematics, to move beyond three dimensions. However, if we try to plot it out, it gets very messy. Fortunately, we can use a good old-fashioned tool—that's a GOFT to those in the know—and make a table of the outcomes. Here's an example that also includes some food for our concert goers. We'll chow on some hotdogs at $5 per hotdog:

$$\text{total\_cost} = 80 \times \text{number\_people} + 10 \times \text{number\_rbs} + 5 \times \text{number\_hotdogs} + 40$$

We'll use a few simple values for the counts of things in our concert-going system:

In [30]:

```
number_people  = np.array([2, 3])
number_rbs     = np.array([0, 1, 2])
number_hotdogs = np.array([2, 4])

costs = np.array([80, 10, 5])

columns = ["People", "RootBeer", "HotDogs", "TotalCost"]
```

I pull off combining several `numpy` arrays in all possible combinations, similar to what `itertools`'s `combinations` function does, with a helper `np_cartesian_product`. It involves a bit of black magic, so I've hidden it in `mlwpy.py`. Feel free to investigate, if you dare.

In [31]:

```
counts = np_cartesian_product(number_people,
                              number_rbs,
                              number_hotdogs)

totals = (costs[0] * counts[:, 0] +
          costs[1] * counts[:, 1] +
          costs[2] * counts[:, 2] + 40)

display(pd.DataFrame(np.c_[counts, totals],
                     columns=columns).head(8))
```

	People	RootBeer	HotDogs	TotalCost
0	2	0	2	210
1	2	0	4	220
2	3	0	2	290
3	3	0	4	300
4	2	1	2	220
5	2	1	4	230
6	3	1	2	300
7	3	1	4	310

The assignment to `totals`—on lines 6–8 in the previous cell—is pretty ugly. Can we improve it? Think! Think! There must be a better way! What is going on there? We are adding several things up. And the things we are adding come from being multiplied together element-wise. Can it be? Is it a dot product? Yes, it is.

In [32]:

```
costs = np.array([80, 10, 5])
counts = np_cartesian_product(number_people,
                              number_rbs,
                              number_hotdogs)

totals = np.dot(counts, costs) + 40
display(pd.DataFrame(np.column_stack([counts, totals]),
                     columns=columns).head(8))
```

	People	RootBeer	HotDogs	TotalCost
0	2	0	2	210
1	2	0	4	220
2	3	0	2	290
3	3	0	4	300
4	2	1	2	220
5	2	1	4	230
6	3	1	2	300
7	3	1	4	310

Using the dot product gets us two wins: (1) the line of code that assigns to `total` is drastically improved and (2) we can more or less arbitrarily extend our costs and counts *without modifying our calculating code at all*. You might notice that I tacked the $+40$ on there by hand. That's because I didn't want to go back to the $+1$ trick—but I could have.

Incidentally, here's what would have happened in a math class. As we saw with the code-line compression from repeated additions to `dot`, details often get abstracted away or moved behind the scenes when we break out advanced notation. Here's a detailed breakdown of what happened. First, we abstract by removing detailed variable names and then replacing our known values by generic identifiers:

$$y = 80x_3 + 10x_2 + 5x_1 + 40$$
$$y = w_3x_3 + w_2x_2 + w_1x_1 + w_0 \cdot 1$$

We take this one step further in code by replacing the wx sums with a dot product:

$$y = w_{[3,2,1]} \bullet x + w_0 \cdot 1$$

The weird $[3, 2, 1]$ subscript on the w indicates that we aren't using *all* of the weights. Namely, we are *not using* the w_0 in the left-hand term. w_0 is in the right-hand term multiplying 1. It is only being used once. The final *coup de grâce* is to perform the $+1$ trick:

$$y = wx^+$$

To summarize, instead of $y = w_3x_3 + w_2x_2 + w_1x_1 + w_0$, we can write $y = wx^+$.

2.7 Notation and the Plus-One Trick

Now that you know what the plus-one trick is, I want to show a few different ways that we can talk about a table of data. That data might be made of values, such as our expense sheet for the trip to the ball park. We can take the table and draw some brackets around it:

$$D = \left(\begin{array}{cc|c} x_2 & x_1 & y \\ 3 & 10 & 3 \\ 2 & 11 & 5 \\ 4 & 12 & 10 \end{array}\right)$$

We can also refer to the parts of it: $D = (\boldsymbol{x}, y)$. Here, $\boldsymbol{x}$ means *all of the input features* and y means the *output target feature*. We can emphasize the columns:

$$D = (\boldsymbol{x}, y) = (x_f, ..., x_1, y)$$

f is the number of features. We're counting backwards to synchronize our weights with the discussion in the prior section. In turn, the weights were backwards so we could count down to the constant term at w_0. It is quite a tangled web.

We can also emphasize the rows:

$$D = \begin{bmatrix} e_1 \\ e_2 \\ \cdots \\ e_n \end{bmatrix}$$

Think of e_i as one example. n is the number of examples.

Also, for mathematical convenience—really—we will often use the augmented versions, the plus-one trick, of D and x:

$$D^+ = (x^+, y) = \left(\begin{array}{ccc|c} x_2 & x_1 & x_0 & y \\ 3 & 10 & 1 & 3 \\ 2 & 11 & 1 & 5 \\ 4 & 12 & 1 & 10 \end{array}\right)$$

Let's break that down:

$$x = \left(\begin{array}{cc} x_2 & x_1 \\ 3 & 10 \\ 2 & 11 \\ 4 & 12 \end{array}\right)$$

If we want to use that with a 2D formula, we end up writing: $y = w_2 x_2 + w_1 x_1 + w_0$. And we can compress that as: $y = w_{[2,1]} \bullet x + w_0$. Again, the $w_{[2,1]}$ is hinting that we aren't using w_0 in the $\bullet$. Still, there is a certain ugliness about the w_0 tacked on at the end. We can compress even further if we use an augmented version of x:

$$x^+ = \left(\begin{array}{ccc} x_2 & x_1 & x_0 \\ 3 & 10 & 1 \\ 2 & 11 & 1 \\ 4 & 12 & 1 \end{array}\right)$$

Now, our 2D formula looks like $y = w_2x_2 + w_1x_1 + w_0x_0$. Note the additional x_0. That fits nicely into $y = w \bullet x^+$, where w is (w_2, w_1, w_0). The augmented version of w now includes w_0, which was previously a weight without a home. When I want to remind you that we are dealing with x^+ or D^+, I'll say *we are using the +1 trick*. We'll connect this mathematical notation to our Python variables in Section 3.3.

2.8 Getting Groovy, Breaking the Straight-Jacket, and Nonlinearity

So, we just took an unsuspecting line and extended it past its comfort zone—maybe past yours as well. We did it in one very specific way: *we added new variables*. These new variables represented new graphical dimensions. We moved from talking about lines to talking about planes and their higher-dimensional cousins.

There is another way in which we can extend the idea of a line. Instead of adding new information—more variables or features—we can add complexity to the information we already have. Imagine moving from $y = 3$ to $y = 2x + 3$ to $y = x^2 + 2x + 3$. In each case, we've added a term to the equation. As we add terms there, we go from a flat line to a sloped line to a parabola. I'll show these off graphically in a second. The key point is: we still only have *one* input variable. We're simply using that single input in different ways.

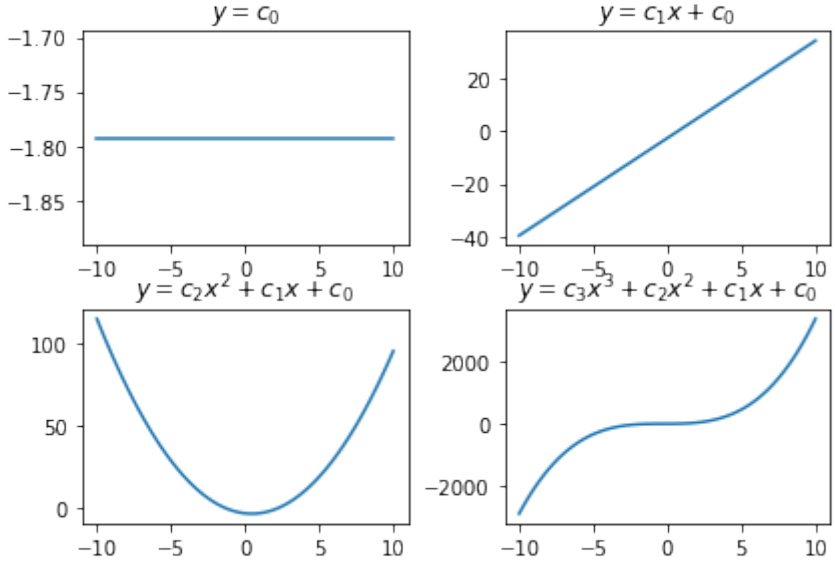
Mathematicians talk about these extensions as adding *higher-order* or *higher-power terms* of the original variable to the equation. As we extend our powers, we get all sorts of fancy names for the functions: constant, linear, quadratic, cubic, quartic, quintic, etc. Usually, we can just call them n-th degree polynomials, where n is the highest non-zero power in the expression. A 2nd degree polynomial—for example, $y = x^2 + x + 1$—is also called a quadratic polynomial. These give us single-bend curves called parabolas.

`np.poly1d` gives us an easy helper to define polynomials by specifying the leading coefficients on each term in the polynomial. For example, we specify $2x^2 + 3x + 4$ by passing in a list of `[2, 3, 4]`. We'll use some random coefficients to get some interesting curves.

In [33]:

```
fig, axes = plt.subplots(2, 2)
fig.tight_layout()

titles = ["$y=c_0$",
          "$y=c_1x+c_0$",
          "$y=c_2x^2+c_1x+c_0$",
          "$y=c_3x^3+c_2x^2+c_1x+c_0$"]

xs = np.linspace(-10, 10, 100)
for power, (ax, title) in enumerate(zip(axes.flat, titles), 1):
    coeffs = np.random.uniform(-5, 5, power)
```

```
    poly = np.poly1d(coeffs)
    ax.plot(xs, poly(xs))
    ax.set_title(title)
```



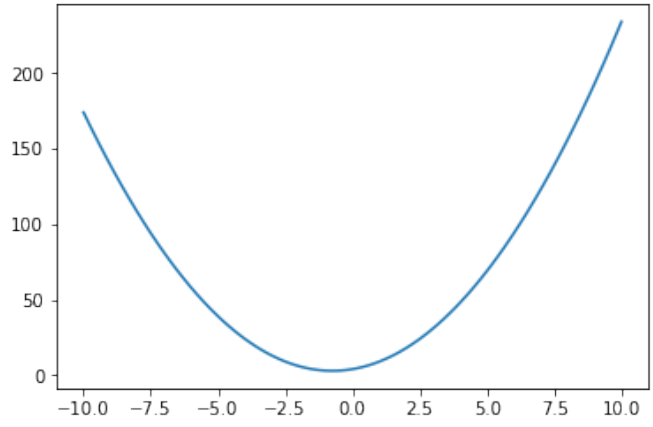
Massaging the general forms of these equations towards our earlier linear equation $y_1 = c_1x + c_0$ gets us to things like $y_2 = c_2x^2 + c_1x + c_0$. One quick note: $x = x^1$ and $1 = x^0$. While I can insert suitable mathese here, trust me that there are very good reasons to define $0^0 = 1$. Taken together, we have

$$y_2 = c_2x^2 + c_1x^1 + c_0x^0 = \sum_{i=0}^{2} c_ix^i$$

You know what I'm about to say. Go ahead, play along and say it with me. You can do it. It's a dot product! We can turn that equation into code by breaking up the x_i and the coefficients c_i and then combining them with a `np.dot`.

In [34]:

```
plt.Figure((2, 1.5))

xs = np.linspace(-10, 10, 101)
coeffs = np.array([2, 3, 4])
ys = np.dot(coeffs, [xs**2, xs**1, xs**0])

# nice parabola via a dot product
plt.plot(xs, ys);
```



2.9 NumPy versus “All the Maths”

Since the dot product is *so fundamental* to machine learning and since NumPy’s `np.dot` has to deal with the practical side of Pythonic computation—as opposed to the pure, Platonic, mathematical world of ideals—I want to spend a few minutes exploring `np.dot` and help you understand how it works in some common cases. More importantly, there is one common form that we’d *like* to use but can’t without some minor adjustments. I want you to know why. Here goes.

We talked about the fact that `np.dot` multiples things element-wise and then adds them up. Here’s just about the most basic example with a 1D array:

In [35]:

```
oned_vec = np.arange(5)
print(oned_vec, "-->", oned_vec * oned_vec)
print("self dot:", np.dot(oned_vec, oned_vec))
```

```
[0 1 2 3 4] --> [ 0  1  4  9 16]
self dot: 30
```

The result is the sum of squares of that array. Here’s a simple example using a row and a column:

In [36]:

```
row_vec = np.arange(5).reshape(1, 5)
col_vec = np.arange(0, 50, 10).reshape(5, 1)
```

Notice that `row_vec` is shaped like a single example and `col_vec` is shaped like a single feature.

```
In [37]:
print("row vec:", row_vec,
      "col_vec:", col_vec,
      "dot:", np.dot(row_vec, col_vec), sep='\n')
```

```
row vec:
[[0 1 2 3 4]]
col_vec:
[[ 0]
 [10]
 [20]
 [30]
 [40]]
dot:
[[300]]
```

So, far, we're mostly good. But what happens if we swap the order? You might expect to get the same answer: after all, in basic arithmetic $3 \times 5 = 5 \times 3$. Let's check it out:

```
In [38]:
out = np.dot(col_vec, row_vec)
print(out)
```

```
[[  0   0   0   0   0]
 [  0  10  20  30  40]
 [  0  20  40  60  80]
 [  0  30  60  90 120]
 [  0  40  80 120 160]]
```

Cue Dorothy: "Toto, I've a feeling we're not in Kansas anymore." What happened here? We'll focus on one output element—the `20` in the second-from-the-top row—to get a handle on the craziness we unleashed. Where does it comes from? Well, we never really defined how the output is produced—except to say that it does a sum product on two 1D arrays. Let's remedy that.

Pick an element in the output, `out[1, 2]`. That's row 1 and column 2, if we start our counting from zero. `out[1, 2]` has the value `20`. Where does this 20 come from? It comes from taking a dot product on row 1 of `col_vec` with column 2 of `row_vec`. That's actually *the definition* of what `np.dot` does. The source values are `col_vec[1,:]` which is `[10]` and `row_vec[:, 2]` which is `[2]`. Putting those together gives $10 \times 2 \rightarrow 20$ with no additional summing needed because we only have one value in each. You can go through a similar process for the other entries.

Mathematically, this is written as $\text{out}_{ij} = \text{dot}(\text{left}_{i\cdot}, \text{right}_{\cdot j})$ where dot is our friendly sum product over 1D things. So, the output row i comes from the left input's row i and the output's column j comes from the right input column j. Taking from each row and each column gives a 5×5 result.

If we apply the same logic to the to the row-column case, we see

In [39]:

```
out = np.dot(row_vec, col_vec)
out
```

Out[39]:

```
array([[300]])
```

The result is 1×1, so `out[0, 0]` comes from row 0 of `row_vec` and column 0 of `col_vec`. Which is exactly the sum product over `[0, 1, 2, 3, 4]` and `[0, 10, 20, 30, 40]`, which gives us `0*0 + 1*10 + 2*20 + 3*30 + 4*40`. Great.

2.9.1 Back to 1D versus 2D

However, when we use a mix of 1D and 2D inputs, things are more confusing because the input arrays are not taken at face value. There are two important consequences for us: (1) the order matters in multiplying a 1D and a 2D array and (2) we have to investigate the rules `np.dot` follows for handling the 1D array.

In [40]:

```
col_vec = np.arange(0, 50, 10).reshape(5, 1)
row_vec = np.arange(0, 5).reshape(1, 5)

oned_vec = np.arange(5)

np.dot(oned_vec, col_vec)
```

Out[40]:

```
array([300])
```

If we trade the order, Python blows up on us:

In [41]:

```
try:
    np.dot(col_vec, oned_vec) # *boom*
except ValueError as e:
    print("I went boom:", e)
```

```
I went boom: shapes (5,1) and (5,) not aligned: 1 (dim 1) != 5 (dim 0)
```

So, `np.dot(oned_vec, col_vec)` works and `np.dot(col_vec, oned_vec)` fails. What’s going on? If we look at the shapes of the guilty parties, we can get a sense of where things break down.

In [42]:

```
print(oned_vec.shape,
      col_vec.shape, sep="\n")
```

```
(5,)
(5, 1)
```

You might consider the following exercise: create a 1D `numpy` array and look at its shape using `.shape`. Transpose it with `.T`. Look at the resulting shape. Take a minute to ponder the mysteries of the NumPy universe. Now repeat with a 2D array. These might not be entirely what you were expecting.

`np.dot` is particular about how these shapes align. Let's look at the row cases:

In [43]:

```
print(np.dot(row_vec, oned_vec))
try: print(np.dot(oned_vec, row_vec))
except: print("boom")
```

```
[30]
boom
```

Here is a summary of what we found:

form	left-input	right-input	success?
`np.dot(oned_vec, col_vec)`	(5,)	(5, 1)	works
`np.dot(col_vec, oned_vec)`	(5, 1)	(5,)	fails
`np.dot(row_vec, oned_vec)`	(1, 5)	(5,)	works
`np.dot(oned_vec, row_vec)`	(5,)	(1, 5)	fails

For the working cases, we can see what happens if we force-reshape the 1D array:

In [44]:

```
print(np.allclose(np.dot(oned_vec.reshape(1, 5), col_vec),
                  np.dot(oned_vec,               col_vec)),
      np.allclose(np.dot(row_vec, oned_vec.reshape(5, 1)),
                  np.dot(row_vec, oned_vec)))
```

```
True True
```

Effectively, for the cases that work, the 1D array is bumped up to $(1, 5)$ if it is on the left and to $(5, 1)$ if it is on the right. Basically, the 1D receives a placeholder dimension on the side it shows up in the `np.dot`. Note that this bumping is *not* using NumPy's full, generic broadcasting mechanism between the two inputs; it is more of a special case.

Broadcasting two arrays against each other in NumPy will result *in the same shape* whether you are broadcasting `a` against `b` or `b` against `a`. Even so, you can *mimic* `np.dot(col_vec, row_vec)` with broadcasting and multiplication. If you do that, you get the "big array" result: it's called an *outer product*.

With all of that said, why do we care? Here's why:

In [45]:

```
D = np.array([[1, 3],
              [2, 5],
              [2, 7],
              [3, 2]])
weights = np.array([1.5, 2.5])
```

This works:

In [46]:

```
np.dot(D,w)
```

Out[46]:

```
array([ -7.5, -12. , -18. ,  -1.5])
```

This fails:

In [47]:

```
try:
    np.dot(w,D)
except ValueError:
    print("BOOM.  :sadface:")
```

```
BOOM.  :sadface:
```

And sometimes, we just want the code to look like our math:

$$y = wD$$

What do we do if we don't like the interface we are given? If we are willing to (1) maintain, (2) support, (3) document, and (4) test an alternative, then we can make an interface that we prefer. Usually people only think about the implementation step. That's a costly mistake.

Here is a version of `dot` that plays nicely with a 1D input as the first argument that is shaped like a column:

In [48]:

```
def rdot(arr,brr):
    'reversed-argument version of np.dot'
    return np.dot(brr,arr)
rdot(w, D)
```

Out[48]:

```
array([ -7.5, -12. , -18. ,  -1.5])
```

You might complain that we are going through contortions to make the code look like the math. That's fair. Even in math textbooks, people will do all sorts of weird gymnastics to make this work: w might be transposed. In NumPy, this is fine, if it is 2D. Unfortunately, if it is only a 1D NumPy array, transposing does *nothing*. Try it yourself! Another gymnastics routine math folks will perform is to transpose the data—that is, they make each feature a row. Yes, really. I'm sorry to be the one to tell you about that. We'll just use `rdot`—short for "reversed arguments to `np.dot`"—when we want our code to match the math.

Dot products are ubiquitous in the mathematics of learning systems. Since we are focused on investigating learning systems through Python programs, it is *really important* that we (1) understand what is going on with `np.dot` and (2) have a convenient and consistent form for using it. We'll see `rdot` in our material on linear and logistic regression. It will also play a role in several other techniques. Finally, it is fundamental in showing the similarities of a wide variety of learning algorithms.

2.10 Floating-Point Issues

Prepare yourself to be grumpy.

In [49]:

```
1.1 + 2.2  == 3.3
```

Out[49]:

```
False
```

I can hear you now. You want your money back—for this book, for your Python program, for *everything*. It's all been a lie. Drama aside, what is happening here? The issue is floating-point numbers and our expectations. In the Python code above, all of the values are `floats`:

In [50]:

```
type(1.1), type(2.2), type(1.1+2.2), type(3.3)
```

Out[50]:

```
(float, float, float, float)
```

`float` is short for *floating-point number*, and floats are how decimal values are usually represented on a computer. When we use floats in our programs, we are often thinking about two different types of numbers: (1) simple decimal values like `2.5` and (2) complicated *real* numbers like π which go on forever, even though we may get away with approximations like 3.14. Both of these have complications when we go from *our* thoughts about these numbers to the *computer's* number-crunching machinery.

Here are a few facts:

1. Computer memory is finite. We can't physically store an infinite number of digits for any numerical value.
2. Some numbers that interest us have an infinite number of decimal places ($\frac{1}{9}$ and π, I'm looking at you).
3. Computers store all of their information in bits—that's base-2 numbers, or *binary*.
4. There are *different* infinite-digit numbers when we write them in decimal versus binary.

Because of points one and two, we have to *approximate* the values we store. We can get *close*, but we can never be *exact*. Because of points three and four, when we convert from a seemingly innocent decimal number like `3.3` to binary, it may become much more complicated—it might have repeating digits, like $\frac{1}{9}$ does in a decimal representation. Putting these pieces together means that *we can't rely on exact comparisons for floating-point values*.

So, what can we do? We can ask if values are *close enough*:

In [51]:

```
np.allclose(1.1 + 2.2, 3.3)
```

Out[51]:

```
True
```

Here, `numpy` is checking if the numbers are the same for many, many decimal places—out to the point where the difference is insignificant. If we care, we can define our own tolerance for what is and isn't significant.

2.11 EOC

2.11.1 Summary

We covered a lot of ideas in this chapter and laid the groundwork to talk about learning in an intelligent way. In many cases, we won't be diving into mathematical details of learning algorithms. However, when we talk about them, we will often appeal to probability,

geometry, and dot products in our descriptions. Hopefully, you now have better intuitions about what these terms and symbols mean—particularly if no one has taken the time to explain them to you in a concrete fashion before.

2.11.2 Notes

While we took an intuitive approach to describing distributions, they have concrete mathematical forms which can be extended to multiple dimensions. The discrete uniform distribution looks like:

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

Here, k is the number of possible events—six for a typical die or two for a coin flip. The equation for the normal distribution is

$$f(x) = \frac{1}{v_m \text{spread}} e^{-\frac{1}{2}(\frac{x-\text{center}}{\text{spread}})^2}$$

The e, combined with a negative power, is responsible for the fast dropoff away from the center. v_m, a *magic value*, is really just there to make sure that all the possibilities sum up to one like all good distributions: it is $v_m = \sqrt{2\pi}$ but I won't quiz you on that. The *center* and *spread* are normally called the *mean* and *standard deviation* and written with μ and σ which are the lowercase Greek *mu* and *sigma*. The normal distribution shows up *everywhere* in statistics: in error functions, in binomial approximations (which we used to generate our normal shapes), and in central limit theorems.

Python uses 0-based indexing while mathematicians often use 1-based indexing. That's because mathematicians are generally *counting* things and computer scientists have historically cared about offsets: from the start, how many steps do I need to move forward to get to the item I need? If I'm at the start of a list or an array, I have to take zero steps to get the first item: I'm already there. A very famous computer scientist, Edsger Dijkstra, wrote an article called "Why numbering should start at zero." Check it out if you are interested and want to win on a computer-science trivia night.

In my mathematical notation, I'm following classic Python's lead and letting both ( ) and [ ] represent ordered things. { } is used for unordered groups of things—imagine putting things in a large duffel bag and then pulling them back out. The duffel bag doesn't remember the order things were put in it. In a relatively recent change, Python dictionaries in Python 3.7 now have some ordering guarantees to them. So, strictly speaking—after upgrading to the latest Python—I'm using the curly braces in the mathematical `set` sense.

The phrase "there must be a better way!"—particularly in the Python community—deserves a hat tip to Raymond Hettinger, a core Python developer. His Python talks are legendary: find them on YouTube and you'll learn something new about Python.

3

Predicting Categories: Getting Started with Classification

In [1]:

```
# setup
from mlwpy import *
%matplotlib inline
```

3.1 Classification Tasks

Now that we've laid a bit of groundwork, let's turn our attention to the main attraction: building and evaluating learning systems. We'll start with classification and we need some data to play with. If that weren't enough, we need to establish some evaluation criteria for success. All of these are just ahead.

Let me squeeze in a few quick notes on terminology. If there are only two target classes for output, we can call a learning task *binary classification*. You can think about $\{Yes, No\}$, $\{Red, Black\}$, or $\{True, False\}$ targets. Very often, binary problems are described mathematically using $\{-1, +1\}$ or $\{0, 1\}$. Computer scientists love to encode $\{False, True\}$ into the numbers $\{0, 1\}$ as the output values. In reality, $\{-1, +1\}$ or $\{0, 1\}$ are both used for mathematical convenience, and it won't make much of a difference to us. (The two encodings often cause head-scratching if you lose focus reading two different mathematical presentations. You might see one in a blog post and the other in an article and you can't reconcile them. I'll be sure to point out any differences in *this* book.) With more than two target classes, we have a *multiclass* problem.

Some classifiers try to make a decision about the output in a direct fashion. The direct approach gives us great flexibility in the relationships we find, but that very flexibility means that we aren't tied down to assumptions that might lead us to better decisions. These assumptions are similar to limiting the suspects in a crime to people that were near where the crime occurred. Sure, we could start with no assumptions at all and equally consider suspects from London, Tokyo, and New York for a crime that occurred in

Nashville. But, adding an assumption that the suspect is in Tennessee should lead to a better pool of suspects.

Other classifiers break the decision into a two-step process: (1) build a model of how likely the outcomes are and (2) pick the most likely outcome. Sometimes we prefer the second approach because we care about the grades of the prediction. For example, we might want to know how likely it is that someone is sick. That is, we want to know that there is a 90% chance someone is sick, versus a more generic estimate "yes, we think they are sick." That becomes important when the real-world cost of our predictions is high. When cost matters, we can combine the probabilities of events with the costs of those events and come up with a decision model to choose a real-world action that balances these, possibly competing, demands. We will consider one example of each type of classifier: Nearest Neighbors goes directly to an output class, while Naive Bayes makes an intermediate stop at an estimated probability.

3.2 A Simple Classification Dataset

The *iris* dataset is included with `sklearn` and it has a long, rich history in machine learning and statistics. It is sometimes called Fisher's Iris Dataset because Sir Ronald Fisher, a mid-20th-century statistician, used it as the sample data in one of the first academic papers that dealt with what we now call classification. Curiously, Edgar Anderson was responsible for gathering the data, but his name is not as frequently associated with the data. Bummer. History aside, what is the *iris* data? Each row describes one iris—that's a flower, by the way—in terms of the length and width of that flower's sepals and petals (Figure 3.1). Those are the big flowery parts and little flowery parts, if you want to be highly technical. So, we have four total measurements per iris. Each of the measurements is a length of one aspect of that iris. The final column, our classification target, is the particular species—one of three—of that iris: *setosa*, *versicolor*, or *virginica*.

We'll load the *iris* data, take a quick tabular look at a few rows, and look at some graphs of the data.

In [2]:

```
iris = datasets.load_iris()

iris_df = pd.DataFrame(iris.data,
                       columns=iris.feature_names)
iris_df['target'] = iris.target
display(pd.concat([iris_df.head(3),
                   iris_df.tail(3)]))
```

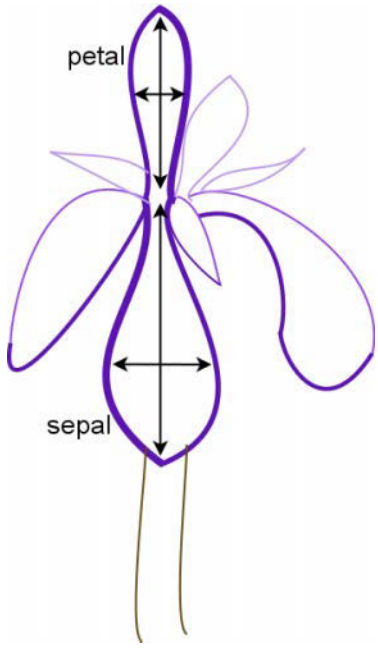
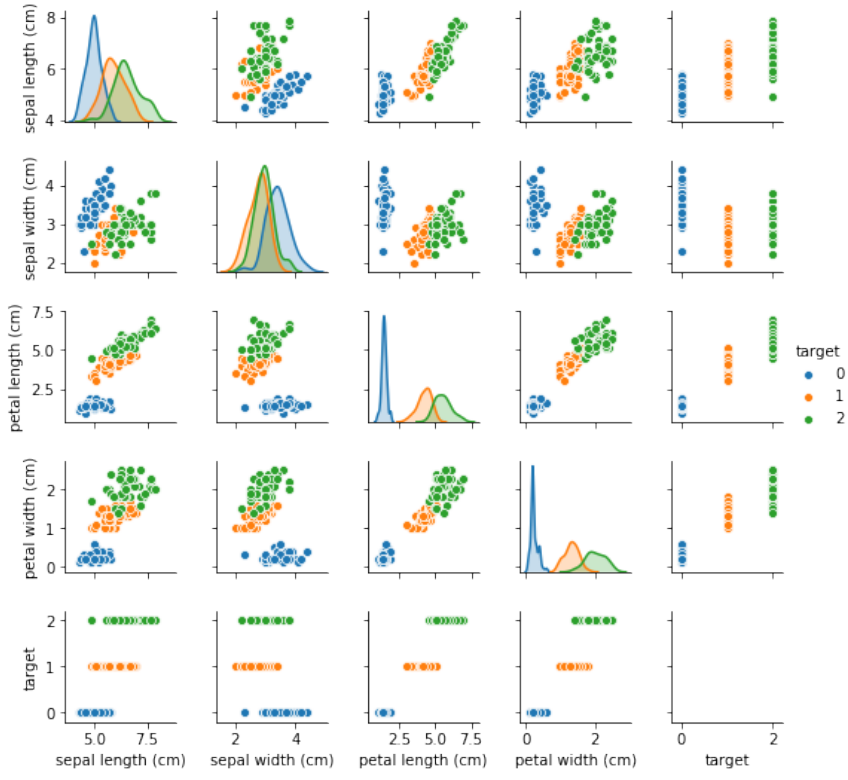


Figure 3.1 An iris and its parts.

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1000	3.5000	1.4000	0.2000	0
1	4.9000	3.0000	1.4000	0.2000	0
2	4.7000	3.2000	1.3000	0.2000	0
147	6.5000	3.0000	5.2000	2.0000	2
148	6.2000	3.4000	5.4000	2.3000	2
149	5.9000	3.0000	5.1000	1.8000	2

In [3]:

```
sns.pairplot(iris_df, hue='target', size=1.5);
```



`sns.pairplot` gives us a nice panel of graphics. Along the diagonal from the top-left to bottom-right corner, we see histograms of the frequency of the different types of iris differentiated by color. The off-diagonal entries—everything *not* on that diagonal—are scatter plots of pairs of features. You'll notice that these pairs occur twice—once above and once below the diagonal—but that each plot for a pair is flipped axis-wise on the other side of the diagonal. For example, near the bottom-right corner, we see *petal width* against *target* and then we see *target* against *petal width* (across the diagonal). When we flip the axes, we change up-down orientation to left-right orientation.

In several of the plots, the blue group (target 0) seems to stand apart from the other two groups. Which species is this?

```
In [4]:
print('targets: {}'.format(iris.target_names),
      iris.target_names[0], sep="\n")
```

```
targets: ['setosa' 'versicolor' 'virginica']
setosa
```

So, looks like *setosa* is easy to separate or partition off from the others. The *v*s, *versicolor* and *virginica*, are more intertwined.

3.3 Training and Testing: Don't Teach to the Test

Let's briefly turn our attention to how we are going to use our data. Imagine you are taking a class (Figure 3.2). Let's go wild and pretend you are studying machine learning. Besides wanting a good grade, when you take a class to learn a subject, you want to be able to use that subject in the real world. Our grade is a surrogate measure for how well we will do in the real world. Yes, I can see your grumpy faces: grades can be very bad estimates of how well we do in the real world. Well, we're in luck! We get to try to make *good* grades that really tell us how well we will do when we get out there to face reality (and, perhaps, our student loans).

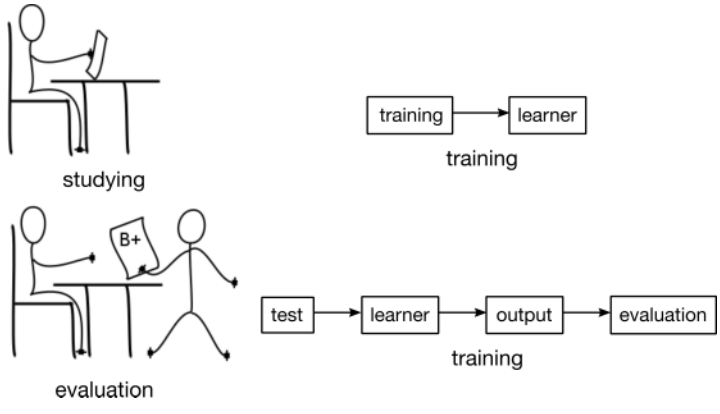


Figure 3.2 School work: training, testing, and evaluating.

So, back to our classroom setting. A common way of evaluating students is to teach them some material and then test them on it. You might be familiar with the phrase "teaching to the test." It is usually regarded as a bad thing. Why? Because, if we teach to the test, the students will do better on the test than on other, new problems they have never seen before. They know the specific answers for the test problems, but they've missed out on the *general* knowledge and techniques they need to answer *novel* problems. Again, remember our goal. We want to do well in the real-world use of our subject. In a machine learning scenario, we want to do well on *unseen* examples. Our performance on unseen examples is called *generalization*. If we test ourselves on data we have already seen, we will have an overinflated estimate of our abilities on novel data.

Teachers prefer to assess students on novel problems. Why? Teachers care about how the students will do on new, never-before-seen problems. If they practice on a specific problem and figure out what's right or wrong about their answer to it, we want that new nugget of knowledge to be something general that they can apply to other problems. If we want to estimate how well the student will do on novel problems, we have to evaluate them on novel problems. Are you starting to feel bad about studying old exams yet?

I don't want to get into too many details of too many tasks here. Still, there is one complication I feel compelled to introduce. Many presentations of learning start off using a teach-to-the-test evaluation scheme called *in-sample evaluation* or *training error*. These have their uses. However, not teaching to the test is such an important concept in learning systems that *I refuse to start you off on the wrong foot!* We just can't take an easy way out. We are going to put on our big girl and big boy pants and do this like adults with a real, *out-of-sample* or *test error* evaluation. We can use these as an estimate for our ability to generalize to unseen, future examples.

Fortunately, `sklearn` gives us some support here. We're going to use a tool from `sklearn` to avoid teaching to the test. The `train_test_split` function segments our dataset that lives in the Python variable `iris`. Remember, that dataset has two components already: the *features* and the *target*. Our new segmentation is going to split it into two buckets of examples:

1. A portion of the data that we will use to study and build up our understanding and
2. A portion of the data that we will use to test ourselves.

We will only study—that is, learn from—the *training* data. To keep ourselves honest, we will only evaluate ourselves on the *testing* data. We promise not to peek at the testing data. We started by breaking our dataset into two parts: features and target. Now, we're breaking each of those into two pieces:

1. Features → training features and testing features
2. Targets → training targets and testing targets

We'll get into more details about `train_test_split` later. Here's what a basic call looks like:

In [5]:

```
# simple train-test split
(iris_train_ftrs, iris_test_ftrs,
 iris_train_tgt,  iris_test_tgt) = skms.train_test_split(iris.data,
                                                         iris.target,
                                                         test_size=.25)
print("Train features shape:", iris_train_ftrs.shape)
print("Test features shape:",  iris_test_ftrs.shape)
```

```
Train features shape: (112, 4)
Test features shape: (38, 4)
```

So, our training data has 112 examples described by four features. Our testing data has 38 examples described by the same four attributes.

If you're confused about the two splits, check out Figure 3.3. Imagine we have a box drawn around a table of our total data. We identify a special column and put that special column on the right-hand side. We draw a vertical line that separates that rightmost column from the rest of the data. That vertical line is the split between our predictive

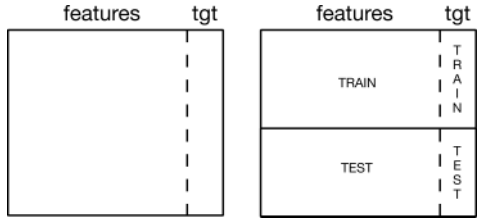


Figure 3.3 Training and testing with features and a target in a table.

features and the target feature. Now, somewhere on the box we draw a horizontal line—maybe three quarters of the way towards the bottom.

The area above the horizontal line represents the part of the data that we use for training. The area below the line is—you got it!—the testing data. And the vertical line? That single, special column is our target feature. In some learning scenarios, there might be multiple target features, but those situations don't fundamentally alter our discussion. Often, we need relatively more data to learn from and we are content with evaluating ourselves on somewhat less data, so the training part might be greater than 50 percent of the data and testing less than 50 percent. Typically, we sort data into training and testing *randomly*: imagine shuffling the examples like a deck of cards and taking the top part for training and the bottom part for testing.

Table 3.1 lists the pieces and how they relate to the *iris* dataset. Notice that I've used both some English phrases and some abbreviations for the different parts. I'll do my best to be consistent with this terminology. You'll find some differences, as you go from book A to blog B and from article C to talk D, in the use of these terms. That isn't the end of the world and there are usually close similarities. Do take a moment, however, to orient yourself when you start following a new discussion of machine learning.

Table 3.1 Relationship between Python variables and *iris* data components.

iris Python variable	Symbol	Phrase
`iris`	D_{all}	(total) dataset
`iris.data`	D_{ftrs}	train and test features
`iris.target`	D_{tgt}	train and test targets
`iris_train_ftrs`	D_{train}	training features
`iris_test_ftrs`	D_{test}	testing features
`iris_train_tgt`	$D_{\text{train}_{\text{tgt}}}$	training target
`iris_test_tgt`	$D_{\text{test}_{\text{tgt}}}$	testing target

One slight hiccup in the table is that `iris.data` refers to all of the input *features*. But this is the terminology that scikit-learn chose. Unfortunately, the Python variable name `data` is sort of like the mathematical x: they are both generic identifiers. `data`, as a name, can refer to just about any body of information. So, while scikit-learn is using a specific sense of the word *data* in `iris.data`, I'm going to use a more specific indicator, D_{ftrs}, for the *features* of the whole dataset.

3.4 Evaluation: Grading the Exam

We've talked a bit about how we want to design our evaluation: we don't teach to the test. So, we train on one set of questions and then evaluate on a new set of questions. How are we going to compute a grade or a score from the exam? For now—and we'll dive into this later—we are simply going to ask, "Is the answer correct?" If the answer is *true* and we predicted *true*, then we get a point! If the answer is *false* and we predicted *true*, we don't get a point. Cue :sadface:. Every correct answer will count as one point. Every missed answer will count as zero points. Every question will count equally for one or zero points. In the end, we want to know the percent we got correct, so we add up the points and divide by the number of questions. This type of evaluation is called *accuracy*, its formula being $\frac{\#\text{correct answers}}{\#\text{questions}}$. It is very much like scoring a multiple-choice exam.

So, let's write a snippet of code that captures this idea. We'll have a very short exam with four true-false questions. We'll imagine a student who finds themself in a bind and, in a last act of desperation, answers every question with `True`. Here's the scenario:

In [6]:

```
answer_key      = np.array([True, True, False, True])
student_answers = np.array([True, True, True, True]) # desperate student!
```

We can calculate the accuracy by hand in three steps:

1. Mark each answer right or wrong.
2. Add up the correct answers.
3. Calculate the percent.

In [7]:

```
correct = answer_key == student_answers
num_correct = correct.sum() # True == 1, add them up
print("manual accuracy:", num_correct / len(answer_key))
```

```
manual accuracy: 0.75
```

Behind the scenes, sklearn's `metrics.accuracy_score` is doing an equivalent calculation:

In [8]:

```
print("sklearn accuracy:",
      metrics.accuracy_score(answer_key,
                             student_answers))
```

```
sklearn accuracy: 0.75
```

So far, we've introduced two key components in our evaluation. First, we identified which material we study from and which material we test from. Second, we decided on a method to score the exam. We are now ready to introduce our first learning method, train it, test it, and evaluate it.

3.5 Simple Classifier #1: Nearest Neighbors, Long Distance Relationships, and Assumptions

One of the simpler ideas for making predictions from a labeled dataset is:

1. Find a way to describe the similarity of two different examples.
2. When you need to make a prediction on a new, unknown example, simply take the value from the most similar known example.

This process is the nearest-neighbors algorithm in a nutshell. I have three friends *Mark*, *Barb*, *Ethan* for whom I know their favorite snacks. A new friend, Andy, is most like Mark. Mark's favorite snack is Cheetos. I predict that Andy's favorite snack is the same as Mark's: Cheetos.

There are many ways we can modify this basic template. We may consider more than *just* the single most similar example:

1. Describe similarity between pairs of examples.
2. Pick several of the most-similar examples.
3. Combine those picks to get a single answer.

3.5.1 Defining Similarity

We have complete control over what *similar* means. We could define it by calculating a *distance* between pairs of examples: `similarity = distance(example_one, example_two)`. Then, our idea of similarity becomes encoded in the way we calculate the distance. Similar things are close—a small distance apart. Dissimilar things are far away—a large distance apart.

Let's look at three ways of calculating the similarity of a pair of examples. The first, *Euclidean* distance, harkens back to high-school geometry or trig. We treat the two examples as points in space. Together, the two points define a line. We let that line be the hypotenuse of a right triangle and, armed with the Pythagorean theorem, use the other two sides of the triangle to calculate a distance (Figure 3.4). You might recall that $c^2 = a^2 + b^2$ or $c = \sqrt{a^2 + b^2}$. Or, you might just recall it as painful. Don't worry, we don't have to *do* the calculation. `scikit-learn` can be told, "Do that *thing* for me." By now, you might be concerned that my next example can only get *worse*. Well, frankly, it could. The *Minkowski* distance would lead us down a path to Einstein and his theory of relativity . . . but we're going to avoid that black (rabbit) hole.

Instead, another option for calculating similarity makes sense when we have examples that consist of simple *Yes*, *No* or *True*, *False* features. With Boolean data, I can compare two examples very nicely by counting up the number of features that are *different*. This simple idea is clever enough that it has a name: the *Hamming* distance. You might recognize this as a close cousin—maybe even a sibling or evil twin—of accuracy. Accuracy is the percent *correct*—the percent of answers the *same* as the target—which is $\frac{\text{correct}}{\text{total}}$. Hamming distance is the number of *differences*. The practical implication is that when two sets of answers agree

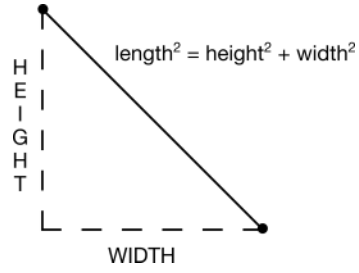


Figure 3.4 Distances from components.

completely, we want the accuracy to be high: 100%. When two sets of features are identical, we want the similarity distance between them to be low: 0.

You might have noticed that these notions of similarity have names—Euclid(-ean), Minkowski, Hamming Distance—that all fit the template of *FamousMathDude Distance*. Aside from the math dude part, the reason they share the term *distance* is because they obey the mathematical rules for what constitutes a distance. They are also called *metrics* by the mathematical wizards-that-be—as in *distance metric* or, informally, a distance measure. These mathematical terms will sometimes slip through in conversation and documentation. `sklearn`'s list of possible distance calculators is in the documentation for `neighbors.DistanceMetric`: there are about twenty metrics defined there.

3.5.2 The *k* in *k*-NN

Choices certainly make our lives complicated. After going to the trouble of choosing how to measure our local neighborhood, we have to decide how to combine the different opinions in the neighborhood. We can think about that as determining who gets to vote and how we will combine those votes.

Instead of considering only *the* nearest neighbor, we might consider some small number of nearby neighbors. Conceptually, expanding our neighborhood gives us more perspectives. From a technical viewpoint, an expanded neighborhood protects us from noise in the data (we'll come back to this in far more detail later). Common numbers of neighbors are 1, 3, 10, or 20. Incidentally, a common name for this technique, and the abbreviation we'll use in this book, is *k-NN* for "*k*-Nearest Neighbors". If we're talking about *k*-NN for classification and need to clarify that, I'll tack a *C* on there: *k-NN-C*.

3.5.3 Answer Combination

We have one last loose end to tie down. We must decide how we combine the known values (votes) from the close, or similar, neighbors. If we have an animal classification problem, four of our nearest neighbors might vote for *cat*, *cat*, *dog*, and *zebra*. How do we respond for our test example? It seems like taking the most frequent response, *cat*, would be a decent method.

In a very cool twist, we can use the exact same neighbor-based technique in *regression* problems where we try to predict a numerical value. The only thing we have to change is how we combine our neighbors' targets. If three of our nearest neighbors gave us numerical values of 3.1, 2.2, and 7.1, how do we combine them? We could use any statistic we wanted, but the mean (average) and the median (middle) are two common and useful choices. We'll come back to *k*-NN for regression in the next chapter.

3.5.4 *k*-NN, Parameters, and Nonparametric Methods

Since *k*-NN is the first model we're discussing, it is a bit difficult to compare it to other methods. We'll save some of those comparisons for later. There's one major difference we can dive into *right now*. I hope that grabbed your attention.

Recall the analogy of a learning model as a machine with knobs and levers on the side. Unlike many other models, *k*-NN outputs—the predictions—can't be computed from an input example and the values of a small, fixed set of adjustable knobs. We need *all* of the training data to figure out our output value. Really? Imagine that we throw out just one of our training examples. That example might be *the* nearest neighbor of a new test example. Surely, missing that training example will affect our output. There are other machine learning methods that have a similar requirement. Still others need some, but not *all*, of the training data when it comes to test time.

Now, you might argue that for a fixed amount of training data there could be a fixed number of knobs: say, 100 examples and 1 knob per example, giving 100 knobs. Fair enough. But then I add one example—and, poof, you now need 101 knobs, and that's a *different* machine. In this sense, the number of knobs on the *k*-NN machine depends on the number of examples in the training data. There is a better way to describe this dependency. Our factory machine had a side tray where we could feed additional information. We can treat the training data as this additional information. Whatever we choose, if we need either (1) a growing number of knobs or (2) the side-input tray, we say the type of machine is *nonparametric*. *k*-NN is a nonparametric learning method.

Nonparametric learning methods can have parameters. (Thank you for nothing, formal definitions.) What's going on here? When we call a method *nonparametric*, it means that with this method, the relationship between features and targets cannot be captured solely using a *fixed* number of parameters. For statisticians, this concept is related to the idea of parametric versus nonparametric statistics: nonparametric statistics assume less about a basket of data. However, recall that we are *not* making any assumptions about the way our black-box factory machine relates to reality. Parametric models (1) make an assumption about the form of the model and then (2) pick a specific model by setting the parameters. This corresponds to the two questions: what knobs are on the machine, and what values are they set to? We don't make assumptions like that with *k*-NN. However, *k*-NN *does* make and rely on assumptions. The most important assumption is that our similarity calculation is related to the *actual* example similarity that we want to capture.

3.5.5 Building a *k*-NN Classification Model

k-NN is our first example of a *model*. Remember, a supervised model is anything that captures the relationship between our features and our target. We need to discuss a few concepts that swirl around the idea of a model, so let's provide a bit of context first. Let's write down a small process we want to walk through:

1. We want to use 3-NN—three nearest neighbors—as our model.
2. We want that model to capture the relationship between the iris training features and the iris training target.
3. We want to use that model to *predict*—on previously unseen test examples—the iris target species.
4. Finally, we want to evaluate the quality of those predictions, using accuracy, by comparing predictions against reality. We didn't peek at these known answers, but we can use them as an answer key for the test.

There's a diagram of the flow of information in Figure 3.5.

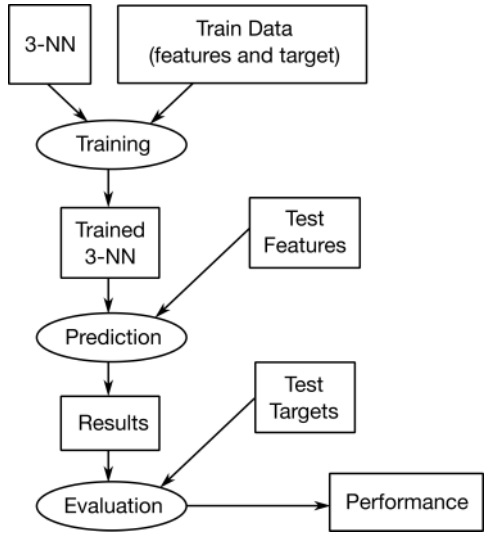


Figure 3.5 Workflow of training, testing, and evaluation for 3-NN.

As an aside on `sklearn`'s terminology, in their documentation an *estimator* is *fit* on some data and then used to *predict* on some data. If we have a training and testing split, we *fit* the *estimator* on *training data* and then use the *fit-estimator* to *predict* on the *test data*. So, let's

1. Create a 3-NN model,
2. Fit that model on the training data,
3. Use that model to predict on the test data, and
4. Evaluate those predictions using accuracy.

In [9]:

```
# default n_neighbors = 5
knn   = neighbors.KNeighborsClassifier(n_neighbors=3)
fit   = knn.fit(iris_train_ftrs, iris_train_tgt)
preds = fit.predict(iris_test_ftrs)

# evaluate our predictions against the held-back testing targets
print("3NN accuracy:",
      metrics.accuracy_score(iris_test_tgt, preds))
```

```
3NN accuracy: 1.0
```

Wow, 100%. We're doing great! This machine learning stuff seems pretty easy—except when it isn't. We'll come back to that shortly. We can abstract away the details of *k*-NN classification and write a simplified workflow template for building and assessing models in `sklearn`:

1. Build the model,
2. Fit the model using the training data,
3. Predict using the fit model on the testing data, and
4. Evaluate the quality of the predictions.

We can connect this workflow back to our conception of a model as a machine. The equivalent steps are:

1. Construct the machine, including its knobs,
2. Adjust the knobs and feed the side-inputs appropriately to capture the training data,
3. Run new examples through the machine to see what the outputs are, and
4. Evaluate the quality of the outputs.

Here's one last, quick note. The `3` in our 3-nearest-neighbors is not something that we adjust by training. It is part of the *internal* machinery of our learning machine. There is no knob on our machine for turning the `3` to a `5`. If we want a 5-NN machine, we have to build a completely different machine. The `3` is not something that is adjusted by the *k*-NN training process. The `3` is a *hyperparameter*. *Hyperparameters* are not trained or manipulated by the learning method they help define. An equivalent scenario is agreeing to the rules of a game and then playing the game under that *fixed* set of rules. Unless we're playing Calvinball or acting like Neo in *The Matrix*—where the flux of the rules is the point—the rules are static for the duration of the game. You can think of hyperparameters as being predetermined and fixed in place before we get a chance to do anything with them while learning. Adjusting them involves conceptually, and literally, working outside the learning box or the factory machine. We'll discuss this topic more in Chapter 11.

3.6 Simple Classifier #2: Naive Bayes, Probability, and Broken Promises

Another basic classification technique that draws directly on probability for its inspiration and operation is the Naive Bayes classifier. To give you insight into the underlying probability ideas, let me start by describing a scenario.

There's a casino that has two tables where you can sit down and play games of chance. At either table, you can play a dice game and a card game. One table is fair and the other table is rigged. Don't fall over in surprise, but we'll call these *Fair* and *Rigged*. If you sit at *Rigged*, the dice you roll have been tweaked and will only come up with six pips—the dots on the dice—one time in ten. The rest of the values are spread equally likely among 1, 2, 3, 4, and 5 pips. If you play cards, the scenario is even worse: the deck at the rigged table has no face cards—kings, queens, or jacks—in it. I've sketched this out in Figure 3.6. For those who want to nitpick, you can't tell these modifications have been made because the dice are visibly identical, the card deck is in an opaque card holder, and you make no physical contact with either the dice or the deck.

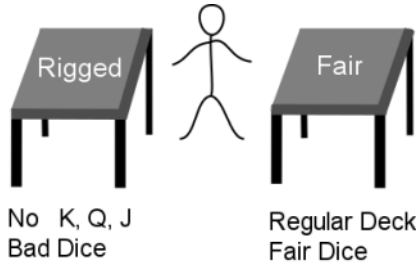


Figure 3.6 Fair and rigged tables at a casino.

Suppose I tell you—truthfully!—that you are sitting at *Rigged*. Then, when you play cards for a while and never see a face card, you aren't surprised. You also won't expect to see sixes on the die very often. Still, if you *know* you are at *Rigged*, neither of the outcomes of the dice or card events is going to *add* anything to your knowledge about the other. We *know* we are at *Rigged*, so *inferring* that we are *Rigged* doesn't add a new fact to our knowledge—although in the real world, confirmation of facts is nice.

Without knowing what table we are at, when we start seeing outcomes we receive information that indicates which table we are at. That can be turned into concrete predictions about the dice and cards. If we *know* which table we're at, that process is short-circuited and we can go directly to predictions about the dice and cards. The information about the table cuts off any gains from seeing a die or card outcome. The story is similar at *Fair*. If I tell you that you just sat down at the fair table, you would expect all the dice rolls to happen with the same probability and the face cards to come up every so often.

Now, imagine you are blindfolded and led to a table. You only know that there are two tables and you know what is happening at both—you know *Rigged* and *Fair* exist.

However, you don't know whether you are at *Rigged* or *Fair*. You sit down and the blindfold is removed. If you are dealt a face card, you immediately know you are at the *Fair* table. When we knew the table we were sitting at, knowing something about the dice didn't tell us anything additional about the cards or vice versa. Now that we don't know the table, we might get some information about the dice from the cards. If we see a face card, which doesn't exist at *Rigged*, we know we *aren't* at *Rigged*. We *must* be at *Fair*. (That's double negative logic put to good use.) As a result, we know that sixes are going to show up regularly.

Our key takeaway is that *there is no communication or causation between the dice and the cards at one of the tables*. Once we sit at *Rigged*, picking a card doesn't adjust the dice odds. The way mathematicians describe this is by saying the cards and the dice are *conditionally independent given the table*.

That scenario lets us discuss the main ideas of Naive Bayes (NB). The key component of NB is that it treats the features as if they are conditionally independent of each other given the class, just like the dice and cards at one of the tables. Knowing the table solidifies our ideas about what dice and cards we'll see. Likewise, knowing a class sets our ideas about what feature values we expect to see.

Since independence of probabilities plays out mathematically as multiplication, we get a very simple description of probabilities in a NB model. The likelihood of features for a given class can be calculated from the training data. From the training data, we store the probabilities of seeing particular features within each target class. For testing, we look up probabilities of feature values associated with a potential target class and multiply them together along with the overall class probability. We do that for each possible class. Then, we choose the class with the highest overall probability.

I constructed the casino scenario to explain what is happening with NB. However, when we use NB as our classification technique, *we assume that the conditional independence between features holds, and then we run calculations on the data*. We could be wrong. The assumptions might be broken! For example, we might not know that every time we roll a specific value on the dice, the dealers—who are *very* good card sharks—are manipulating the deck we draw from. If that were the case, there *would* be a connection between the deck and dice; our assumption that there is no connection would be *wrong*. To quote a famous statistician, George Box, "All models are wrong but some are useful." Indeed.

Naive Bayes can be *very* useful. It turns out to be unreasonably useful in text classification. This is almost mind-blowing. It seems obvious that the words in a sentence depend on each other and on their order. We don't pick words at random; we intentionally put the right words together, in the right order, to communicate specific ideas. How can a method which *ignores* the relationship between words—which are the basis of our features in text classification—be so useful? The reasoning behind NB's success is two-fold. First, Naive Bayes is a relatively *simple* learning method that is hard to distract with irrelevant details. Second, since it is particularly simple, it benefits from having *lots* of data fed into it. I'm being slightly vague here, but you'll need to jump ahead to the discussion of *overfitting* (Section 5.3) to get more out of me.

Let's build, fit, and evaluate a simple NB model.

In [10]:

```
nb    = naive_bayes.GaussianNB()
fit   = nb.fit(iris_train_ftrs, iris_train_tgt)
preds = fit.predict(iris_test_ftrs)

print("NB accuracy:",
      metrics.accuracy_score(iris_test_tgt, preds))
```

```
NB accuracy: 1.0
```

Again, we are perfect. Don't be misled, though. Our success says more about the ease of the dataset than our skills at machine learning.

3.7 Simplistic Evaluation of Classifiers

We have everything lined up for the fireworks! We have data, we have methods, and we have an evaluation scheme. As the Italians say, "*Andiamo!*" Let's go!

3.7.1 Learning Performance

Shortly, we'll see a simple Python program to compare our two learners: *k*-NN and NB. Instead of using the names imported by our setup statement `from mlwpy import *` at the start of the chapter, it has its `import`s written out. This code is what you would write in a stand-alone script or in a notebook that *doesn't* import our convenience setup. You'll notice that we rewrote the `train_test_split` call and we also made the test set size significantly bigger. Why? Training on less data makes it a harder problem. You'll also notice that I sent an extra argument to `train_test_split`: `random_state=42` hacks the randomness of the train-test split and gives us a repeatable result. Without it, every run of the cell would result in different evaluations. Normally we want that, but here I want to be able to talk about the results *knowing* what they are.

In [11]:

```
# stand-alone code
from sklearn import (datasets, metrics,
                     model_selection as skms,
                     naive_bayes, neighbors)

# we set random_state so the results are reproducible
# otherwise, we get different training and testing sets
# more details in Chapter 5
iris = datasets.load_iris()
```

```
(iris_train_ftrs, iris_test_ftrs,
 iris_train_tgt, iris_test_tgt) = skms.train_test_split(iris.data,
                                                        iris.target,
                                                        test_size=.90,
                                                        random_state=42)

models = {'kNN': neighbors.KNeighborsClassifier(n_neighbors=3),
          'NB' : naive_bayes.GaussianNB()}

for name, model in models.items():
    fit = model.fit(iris_train_ftrs, iris_train_tgt)
    predictions = fit.predict(iris_test_ftrs)

    score = metrics.accuracy_score(iris_test_tgt, predictions)
    print("{:>3s}: {:0.2f}".format(name,score))
```

```
kNN: 0.96
 NB: 0.81
```

With a test set size of 90% of the data, *k*-NN does fairly well and NB does a bit *meh* on this train-test split. If you rerun this code many times without `random_state` set and you use a more moderate amount of testing data, we get upwards of 97+% accuracy on both methods for many repeated runs. So, from a learning performance perspective, *iris* is a fairly easy problem. It is reasonably easy to distinguish the different types of flowers, based on the measurements we have, using very simple classifiers.

3.7.2 Resource Utilization in Classification

Everything we do on a computer comes with a cost in terms of processing time and memory. Often, computer scientists will talk about memory as storage space or, simply, space. Thus, we talk about the *time and space* usage of a program or an algorithm. It may seem a bit old-fashioned to worry about resource usage on a computer; today's computer are orders of magnitude faster and larger in processing and storage capabilities than their ancestors of even a few years ago—let alone the behemoth machines of the 1960s and 1970s. So why are we going down a potentially diverting rabbit hole? There are two major reasons: extrapolation and the limits of theoretical analysis.

3.7.2.1 Extrapolation

Today, much of data science and machine learning is driven by *big data*. The very nature of big data is that it pushes the limits of our computational resources. Big data is a relative term: what's big for you might not be too big for someone with the skills and budget to compute on a large cluster of machines with GPUs (graphics processing units). One possible breaking point after which I *don't* have *small* data is when the problem is so large that I can't solve it on my laptop in a "reasonable" amount of time.

If I'm doing my prototyping and development on my laptop—so I can sip a mojito under a palm tree in the Caribbean while I'm working—how can I know what sort of resources I will need when I scale up to the full-sized problem? Well, I can take measurements of smaller problems of increasing sizes and make some educated guesses about what will happen with the full dataset. To do that, I need to quantify what's happening with the smaller data in time and space. In fairness, it is only an estimate, and adding computational horsepower doesn't always get a one-to-one payback. Doubling my available memory won't always double the size of the dataset I can process.

3.7.2.2 Limits of Theory

Some of you might be aware of a subfield of computer science called *algorithm analysis* whose job is to develop equations that relate the time and memory use of a computing task to the size of that task's input. For example, we might say that the new learning method *Foo* will take $2n + 27$ steps on n input examples. (That's a drastic simplification: we almost certainly care about how many features there are in these examples.)

So, if there is a theoretical way to know the resources needed by an algorithm, why do we care about measuring them? I'm glad you asked. Algorithm analysis typically abstracts away certain mathematical details, like constant factors and terms, that can be practically relevant to real-world run times. Algorithm analysis also (1) makes certain strong or mathematically convenient assumptions, particularly regarding the average case analysis, (2) can ignore implementation details like system architecture, and (3) often uses algorithmic idealizations, devoid of real-world practicalities and necessities, to reach its conclusions.

In short, the only way to *know* how a real-world computational system is going to consume resources, short of some specialized cases that don't apply here, is to run it and measure it. Now, it is just as possible to screw this up: you could run and measure under idealized or nonrealistic conditions. We don't want to throw out algorithmic analysis altogether. My critiques are *not* failures of algorithm analysis; it's simply open-eyed understanding its limits. Algorithm analysis will always tell us some fundamental truths about how different algorithms compare and how they behave on bigger-and-bigger inputs.

I'd like to show off a few methods of comparing the resource utilization of our two classifiers. A few caveats: quantifying program behavior can be very difficult. Everything occurring on your system can potentially have a significant impact on your learning system's resource utilization. Every difference in your input can affect your system's behavior: more examples, more features, different types of features (numerical versus symbolic), and different hyperparameters can all make the same learning algorithm behave differently and consume different resources.

3.7.2.3 Units of Measure

We need to make one small digression. We're going to be measuring the resources used by computer programs. Time is measured in seconds, and space is measured in bytes. One byte is eight bits: it can hold the answers to eight yes/no questions. Eight bits can

distinguish between 256 different values—so far, so good. However, we'll be dealing with values that are significantly larger or smaller than our normal experience. I want you to be able to connect with these values.

We need to deal with SI prefixes. SI is short for the International Standard of scientific abbreviations—but, coming from a Romance language, the adjective is *after* the noun, so the IS is swapped. The prefixes that are important for us are in Table 3.2. Remember that the exponent is the x in 10^x; it's also the number of "padded zeros" on the right. That is, *kilo* means $10^3 = 1000$ and 1000 has three zeros on the right. The examples are distances that would be reasonable to measure, using that prefix, applied to meters.

Table 3.2 SI prefixes and length scale examples.

Prefix	Verbal	Exponent	Example Distance
T	tera	12	orbit of Neptune around the Sun
G	giga	9	orbit of the Moon around the Earth
M	mega	6	diameter of the Moon
K	kilo	3	a nice walk
		0	1 meter $\sim$ 1 step
m	milli	-3	mosquito
μ	micro	-6	bacteria
n	nano	-9	DNA

There is another complicating factor. Computers typically work with base-2 amounts of storage, not base-10. So, instead of 10^x we deal with 2^x. Strictly speaking—and scientists are nothing if not strict—we need to account for this difference. For memory, we have some additional prefixes (Table 3.3) that you'll see in use soon.

Table 3.3 SI base-two prefixes and memory scale examples.

Prefix	Verbal Prefix	Number of Bytes	Example
KiB	kibi	2^{10}	a list of about 1000 numbers
MiB	mebi	2^{20}	a short song as an MP3
GiB	gibi	2^{30}	a feature-length movie
TiB	tebi	2^{40}	a family archive of photos and movies

So, 2 MiB is *two mebi-bytes* equal to 2^{20} bytes. You'll notice that the base-2 prefixes are also pronounced differently. Ugh. You might wonder why these step up by 10s, not by 3s as in the base-10 values. Since $2^{10} = 1024 \sim 1000 = 10^3$, multiplying by ten 2s is fairly close to multiplying by three 10s. Unfortunately, these binary prefixes, defined by large standards bodies, haven't necessarily trickled down to daily conversational use. The good news is that within one measuring system, you'll probably only see MiB or MB, not both. When you see MiB, just know that it isn't quite MB.

3.7.2.4 Time

In a Jupyter notebook, we have some nice tools to measure execution times. These are great for measuring the time use of small snippets of code. If we have two different ways of coding a solution to a problem and want to compare their speed, or just want to measure how long a snippet of code takes, we can use Python's `timeit` module. The Jupyter cell magic `%timeit` gives us a convenient interface to time a line of code:

```
In [12]:
```

```
%timeit -r1 datasets.load_iris()
```

```
1000 loops, best of 1: 1.4 ms per loop
```

The `-r1` tells `timeit` to measure the timing of the snippet once. If we give a higher `r`, for repeats, the code will be run multiple times and we will get statistics. Recent versions of Jupyter default to calculating the mean and standard deviation of the results. Fortunately, for a single result we just get that single value. If you are concerned about the `1000 loops`, check out my note on it at the end of the chapter.

`%%timeit`—the two-percents make it a *cell magic*—applies the same strategy to the entire block of code in a cell:

```
In [13]:
```

```
%%timeit -r1 -n1
(iris_train_ftrs, iris_test_ftrs,
 iris_train_tgt,  iris_test_tgt) = skms.train_test_split(iris.data,
                                                         iris.target,
                                                         test_size=.25)
```

```
1 loop, best of 1: 638 µs per loop
```

And now let's point our chronometer (`timeit`) at our learning workflow:

```
In [14]:
```

```
%%timeit -r1

nb    = naive_bayes.GaussianNB()
fit   = nb.fit(iris_train_ftrs, iris_train_tgt)
preds = fit.predict(iris_test_ftrs)

metrics.accuracy_score(iris_test_tgt, preds)
```

```
1000 loops, best of 1: 1.07 ms per loop
```

In [15]:

```
%%timeit -r1

knn   = neighbors.KNeighborsClassifier(n_neighbors=3)
fit   = knn.fit(iris_train_ftrs, iris_train_tgt)
preds = fit.predict(iris_test_ftrs)

metrics.accuracy_score(iris_test_tgt, preds)
```

```
1000 loops, best of 1: 1.3 ms per loop
```

If we just want to time one line in a cell—for example, we only want to see how long it takes to fit the models—we can use a single-percent version, called a *line magic*, of `timeit`:

In [16]:

```
# fitting
nb = naive_bayes.GaussianNB()
%timeit -r1 fit   = nb.fit(iris_train_ftrs, iris_train_tgt)

knn = neighbors.KNeighborsClassifier(n_neighbors=3)
%timeit -r1 fit = knn.fit(iris_train_ftrs, iris_train_tgt)
```

```
1000 loops, best of 1: 708 µs per loop
1000 loops, best of 1: 425 µs per loop
```

In [17]:

```
# predicting
nb    = naive_bayes.GaussianNB()
fit   = nb.fit(iris_train_ftrs, iris_train_tgt)
%timeit -r1 preds = fit.predict(iris_test_ftrs)

knn   = neighbors.KNeighborsClassifier(n_neighbors=3)
fit   = knn.fit(iris_train_ftrs, iris_train_tgt)
%timeit -r1 preds = fit.predict(iris_test_ftrs)
```

```
1000 loops, best of 1: 244 µs per loop
1000 loops, best of 1: 644 µs per loop
```

There seems to be a bit of a tradeoff. *k*-NN is faster to fit, but is slower to predict. Conversely, NB takes a bit of time to fit, but is faster predicting. If you're wondering why I didn't reuse the `knn` and `nb` from the prior cell, it's because when you `%timeit`, variable assignment are trapped inside the `timeit` magic and don't leak back out to our main code. For example, trying to use `preds` as "normal" code in the prior cell will results in a `NameError`.

3.7.2.5 Memory

We can also do a very similar sequence of steps for quick-and-dirty measurements of memory use. However, two issues raise their ugly heads: (1) our tool isn't built into Jupyter, so we need to install it and (2) there are technical details—err, opportunities?—that we'll get to in a moment. As far as installation goes, install the `memory_profiler` module with `pip` or `conda` at your terminal command line:

```
pip install memory_profiler
conda install memory_profiler
```

Then, in your notebook you will be able to use `%load_ext`. This is Jupyter's command to load a Jupyter extension module—sort of like Python's `import`. For `memory_profiler`, we use it like this:

```
%load_ext memory_profiler
```

Here it goes:

```
In [18]:
```

```
%load_ext memory_profiler
```

Use it is just like `%%timeit`. Here's the cell magic version for Naive Bayes:

```
In [19]:
```

```
%%memit
nb    = naive_bayes.GaussianNB()
fit   = nb.fit(iris_train_ftrs, iris_train_tgt)
preds = fit.predict(iris_test_ftrs)
```

```
peak memory: 144.79 MiB, increment: 0.05 MiB
```

And for Nearest Neighbors:

```
In [20]:
```

```
%%memit
knn   = neighbors.KNeighborsClassifier(n_neighbors=3)
fit   = knn.fit(iris_train_ftrs, iris_train_tgt)
preds = fit.predict(iris_test_ftrs)
```

```
peak memory: 144.79 MiB, increment: 0.00 MiB
```

3.7.2.6 Complicating Factors

You may never have considered what happens with memory on your computer. In the late 2010s, you might have 4 or 8GB of system memory, RAM, on your laptop. I have 32GB

on my workhorse powerstation—or workstation powerhorse, if you prefer. Regardless, that system memory is shared by each and every running program on your computer. It is the job of the operating system—Windows, OSX, Linux are common culprits—to manage that memory and respond to applications' requests to use it. The OS has to be a bit of a playground supervisor to enforce sharing between the different programs.

Our small Python programs, too, are playing on that playground. We have to share with others. As we request resources like memory—or time on the playground swing—the OS will respond and give us a block of memory to use. We might actually get *more* memory than we request (more on that in a second). Likewise, when we are done with a block of memory—and being the polite playground children that we are—we will return it to the playground monitor. In both our request for memory and our return of the memory, the process incurs management overhead. Two ways that OSes simplify the process and reduce the overhead are (1) by granting memory in blocks that might be more than we need and (2) by possibly letting us keep using memory, after we've said we're done with it, until someone else *actively* needs it. The net result of this is that determining the actual amount of memory that we are using—versus the amount the operating system has walled off for us—can be very tricky. Measuring additional requests within a running program is even more difficult.

Another issue further complicates matters. Python is a memory-managed language: it has its own memory management facilities on top of the OS. If you were to rerun the above cells in a Jupyter notebook, you might see a memory increment of 0.00 MiB and wonder what circuits just got fried. In that case, the old memory we used was released by us—and the operating system never shuffled it off to someone else. So, when we needed more memory, we were able to reuse the old memory and didn't need any new memory from the OS. It is almost as if the memory was released and reclaimed by us so quickly that it was never actually gone! Now, whether or not we see an increment is also dependent on (1) what the notebook cell is doing, (2) what other memory our program has claimed and is using, (3) every other program that is running on the computer, and (4) the exact details of the operating system's memory manager. To learn more, check out a course or textbook on operating systems.

3.7.3 Stand-Alone Resource Evaluation

To minimize these concerns and to reduce confounding variables, it is extremely useful to write small, stand-alone programs when testing memory use. We can make the script general enough to be useful for stand-alone timing, as well.

In [21]:

```
!cat scripts/knn_memtest.py
```

```
import memory_profiler, sys
from mlwpy import *

@memory_profiler.profile(precision=4)
```

```
def knn_memtest(train, train_tgt, test):
    knn   = neighbors.KNeighborsClassifier(n_neighbors=3)
    fit   = knn.fit(train, train_tgt)
    preds = fit.predict(test)

if __name__ == "__main__":
    iris = datasets.load_iris()
    tts = skms.train_test_split(iris.data,
                                iris.target,
                                test_size=.25)
    (iris_train_ftrs, iris_test_ftrs,
     iris_train_tgt,  iris_test_tgt) = tts
    tup = (iris_train_ftrs, iris_train_tgt, iris_test_ftrs)
    knn_memtest(*tup)
```

There are a few ways to use `memory_profiler`. We've seen the line and cell magics in the previous section. In `knn_memtest.py`, we use the `@memory_profiler.profile` *decorator*. That extra line of Python tells the memory profiler to track the memory usage of `knn_memtest` on a line-by-line basis. When we run the script, we see memory-related output for each line of `knn_memtest`:

```
In [22]:
!python scripts/knn_memtest.py
```

```
Filename: scripts/knn_memtest.py
# output modified for formatting purposes

Line #    Mem usage    Increment   Line Contents
================================================
     4 120.5430 MiB 120.5430 MiB   @memory_profiler.profile(precision=4)
     5                             def knn_memtest(train, train_tgt, test):
     6 120.5430 MiB   0.0000 MiB       knn   = neighbors.
                                           KNeighborsClassifier(n_neighbors=3)
     7 120.7188 MiB   0.1758 MiB       fit   = knn.fit(train, train_tgt)
     8 120.8125 MiB   0.0938 MiB       preds = fit.predict(test)
```

Here's another stand-alone script to measure the memory usage of Naive Bayes:

```
In [23]:
import functools as ft
import memory_profiler
from mlwpy import *

def nb_go(train_ftrs, test_ftrs, train_tgt):
    nb    = naive_bayes.GaussianNB()
```

```
    fit   = nb.fit(train_ftrs, train_tgt)
    preds = fit.predict(test_ftrs)

def split_data(dataset):
    split = skms.train_test_split(dataset.data,
                                  dataset.target,
                                  test_size=.25)
    return split[:-1] # don't need test tgt

def msr_mem(go, args):
    base = memory_profiler.memory_usage()[0]
    mu = memory_profiler.memory_usage((go, args),
                                      max_usage=True)[0]
    print("{:<3}: ~{:.4f} MiB".format(go.__name__, mu-base))

if __name__ == "__main__":
    msr = msr_mem
    go = nb_go

    sd = split_data(datasets.load_iris())
    msr(go, sd)
```

```
nb_go: ~0.0078 MiB
```

`nb_go` has the *model-fit-predict* pattern we saw above. `split_data` just wraps `train_test_split` in a convenient way to use with `nb_go`. The new piece is setting up the timing wrapper in `msr_mem`. Essentially, we ask what memory is used now, run `nb_go`, and then see the maximum memory used along the way. Then, we take that max, subtract what we were using before, `max-baseline`, and that's the peak memory used by `nb_go`. `nb_go` gets passed in to `msr_mem` as `go` and then finds its way to `memory_usage`.

We can write a similar `msr_time` driver to evaluate time, and we can write a similar `knn_go` to kick off a *k*-NN classifier for measuring time and memory. Here are all four pieces in a single script:

In [24]:

```
!cat scripts/perf_01.py
```

```
import timeit, sys
import functools as ft
import memory_profiler
from mlwpy import *

def knn_go(train_ftrs, test_ftrs, train_tgt):
    knn   = neighbors.KNeighborsClassifier(n_neighbors=3)
    fit   = knn.fit(train_ftrs, train_tgt)
```

```
    preds = fit.predict(test_ftrs)

def nb_go(train_ftrs, test_ftrs, train_tgt):
    nb    = naive_bayes.GaussianNB()
    fit   = nb.fit(train_ftrs, train_tgt)
    preds = fit.predict(test_ftrs)

def split_data(dataset):
    split = skms.train_test_split(dataset.data,
                                  dataset.target,
                                  test_size=.25)
    return split[:-1] # don't need test tgt

def msr_time(go, args):
    call = ft.partial(go, *args)
    tu = min(timeit.Timer(call).repeat(repeat=3, number=100))
    print("{:<6}: ~{:.4f} sec".format(go.__name__, tu))

def msr_mem(go, args):
    base = memory_profiler.memory_usage()[0]
    mu = memory_profiler.memory_usage((go, args),
                                      max_usage=True)[0]
    print("{:<3}: ~{:.4f} MiB".format(go.__name__, mu-base))

if __name__ == "__main__":
    which_msr = sys.argv[1]
    which_go = sys.argv[2]

    msr = {'time': msr_time, 'mem':msr_mem}[which_msr]
    go = {'nb' : nb_go, 'knn': knn_go}[which_go]

    sd = split_data(datasets.load_iris())
    msr(go, sd)
```

With all this excitement, let's see where we end up using Naive Bayes:

In [25]:

```
!python scripts/perf_01.py mem nb
!python scripts/perf_01.py time nb
```

```
nb_go: ~0.1445 MiB
nb_go : ~0.1004 sec
```

And with *k*-NN:

In [26]:

```
!python scripts/perf_01.py mem knn
!python scripts/perf_01.py time knn
```

```
knn_go: ~0.3906 MiB
knn_go: ~0.1035 sec
```

In summary, our learning and resource performance metrics look like this (the numbers may vary a bit):

Method	Accuracy	~Time(s)	~Memory (MiB)
k-NN	0.96	0.10	.40
NB	0.80	0.10	.14

Don't read too much into the accuracy scores! I'll tell you why in a minute.

3.8 EOC

3.8.1 Sophomore Warning: Limitations and Open Issues

There are several caveats to what we've done in this chapter:

- We compared these learners on a single dataset.
- We used a very simple dataset.
- We did *no* preprocessing on the dataset.
- We used a single train-test split.
- We used accuracy to evaluate the performance.
- We didn't try different numbers of neighbors.
- We only compared two simple models.

Each one of these caveats is great! It means we have more to talk about in the forthcoming chapters. In fact, discussing *why* these are concerns and figuring out *how* to address them is the point of this book. Some of these issues have no fixed answer. For example, no one learner is best on *all* datasets. So, to find a good learner *for a particular problem*, we often try several different learners and pick the one that does the best *on that particular problem*. If that sounds like teaching-to-the-test, you're right! We have to be very careful in how we select the model we use from many potential models. Some of these issues, like our use of accuracy, will spawn a long discussion of how we quantify and visualize the performance of classifiers.

3.8.2 Summary

Wrapping up our discussion, we've seen several things in this chapter:

1. *iris*, a simple real-world dataset
2. Nearest-neighbors and Naive Bayes classifiers
3. The concept of training and testing data
4. Measuring learning performance with accuracy
5. Measuring time and space usage within a Jupyter notebook and via stand-alone scripts

3.8.3 Notes

If you happen to be a botanist or are otherwise curious, you can read Anderson's original paper on irises: www.jstor.org/stable/2394164. The version of the *iris* data with `sklearn` comes from the UCI Data repository: https://archive.ics.uci.edu/ml/datasets/iris.

The Minkowski distance isn't really as scary as it seems. There's another distance called the Manhattan distance. It is the distance it would take to walk as directly as possible from one point to the other, if we were on a fixed grid of streets like in Manhattan. It simply adds up the absolute values of the feature differences without squares or square roots. All Minkowski does is extend the formulas so we can pick Manhattan, Euclidean, or other distances by varying a value p. The weirdness comes in when we make p very, very big: $p \to \infty$. Of course, that has its own name: the Chebyshev distance.

If you've seen theoretical resource analysis of algorithms before, you might remember the terms *complexity analysis* or *Big-O* notation. The Big-O analysis simplifies statements on the upper bounds of resource use, as input size grows, with mathematical statements like $\mathcal{O}(n^2)$—hence the name Big-O.

I briefly mentioned graphics processing units (GPUs). When you look at the mathematics of computer graphics, like the visuals in modern video games, it is all about describing points in space. And when we play with data, we often talk about examples as points in space. The "natural" mathematical language to describe this is *matrix algebra*. GPUs are designed to perform matrix algebra at warp speed. So, it turns out that machine learning algorithms can be run very, very efficiently on GPUs. Modern projects like Theano, TensorFlow, and Keras are designed to take advantage of GPUs for learning tasks, often using a type of learning model called a *neural network*. We'll briefly introduce these in Chapter 15.

In this chapter, we used Naive Bayes on discrete data. Therefore, learning involved making a table of how often values occurred for the different target classes. When we have continuous numerical values, the game is a bit different. In that case, learning means figuring out the center and spread of a distribution of values. Often, we assume that a *normal* distribution works well with the data; the process is then called *Gaussian Naive Bayes*—Gaussian and normal are essentially synonyms. Note that we are making an *assumption*—it might work well but we might also be *wrong*. We'll talk more about GNB in Section 8.5.

In any chapter that discusses performance, I would be remiss if I didn't tell you that "premature optimization is the root of all evil . . . in programming." This quote is from an essay form of Donald Knuth's 1974 Turing Award—the Nobel Prize of Computer Science—acceptance speech. Knuth is, needless to say, a giant in the discipline. There are two points that underlie his quote. Point one: in a computer system, the majority of the execution time is usually tied up in a small part of the code. This observation is a form of the Pareto principle or the 80–20 rule. Point two: optimizing code is hard, error-prone, and makes the code more difficult to understand, maintain, and adapt. Putting these two points together tells us that we can waste an awful lot of programmer time optimizing code that isn't contributing to the overall performance of our system. So, what's the better way? (1) Write a good, solid, *working* system and then measure its performance. (2) Find the bottlenecks—the slow and/or calculation-intensive portions of the program. (3) Optimize those bottlenecks. We only do the work that we *know* needs to be done and has a chance at meeting our goals. We also do as little of this intense work as possible. One note: *inner loops*—the innermost nestings of repetition—are often the most fruitful targets for optimization because they are, by definition, code that is repeated the most times.

Recent versions of Jupyter now report a mean and standard deviation for `%timeit` results. However, the Python core developers and documenters prefer a different strategy for analyzing `timeit` results: they prefer either (1) taking the minimum of several repeated runs to give an idea of best-case performance, which will be more consistent for comparison sake, or (2) looking at all of the results as a whole, without summary. I think that (2) is *always* a good idea in data analysis. The mean and standard deviation are not *robust*; they respond poorly to outliers. Also, while the mean and standard deviation completely characterize normally distributed data, other distributions will be characterized in very different ways; see Chebyshev's inequality for details. I would be far happier if Jupyter reported medians and inter-quartile ranges (those are the 50th percentile and the 75th–25th percentiles). These are robust to outliers and are not based on distributional assumptions about the data.

What was up with the `1000 loops` in the `timeit` results? Essentially, we are stacking multiple runs of the same, potentially short-lived, task one after the other so we get a longer-running pseudo-task. This longer-running task plays more nicely with the level of detail that the timing functions of the operating system support. Imagine measuring a 100-yard dash using a sundial. It's going to be very hard because there's a mismatch between the time scales. As we repeat the task multiple times—our poor sprinters might get worn out but, fortunately, Python keeps chugging along—we may get more meaningful measurements. Without specifying a `number`, `timeit` will attempt to find a good number for you. In turn, this may take a while because it will try increasing values for `number`. There's also a `repeat` value you can use with `timeit`; `repeat` is an *outer loop* around the whole process. That's what we discussed computing statistics on in the prior paragraph.

3.8.4 Exercises

You might be interested in trying some classification problems on your own. You can follow the model of the sample code in this chapter with some other classification datasets

from `sklearn`: `datasets.load_wine` and `datasets.load_breast_cancer` will get you started. You can also download numerous datasets from online resources like:

- The UCI Machine Learning Repository, https://archive.ics.uci.edu/ml/datasets.html
- Kaggle, www.kaggle.com/datasets

4

Predicting Numerical Values: Getting Started with Regression

In [1]:

```
# setup
from mlwpy import *
%matplotlib inline
```

4.1 A Simple Regression Dataset

Regression is the process of predicting a finely graded numerical value from inputs. To illustrate, we need a simple dataset that has numerical results. `sklearn` comes with the *diabetes* dataset that will serve us nicely. The dataset consists of several biometric and demographic measurements. The version included with `sklearn` has been modified from raw numerical features by subtracting the mean and dividing by the standard deviation of each column. That process is called *standardizing* or *z-scoring* the features. We'll return to the standard deviation later; briefly, it is a measure of how spread out a set of values are.

The net result of standardizing the columns is that each column has a mean of 0 and a standard deviation of 1. We standardize, or otherwise rescale, the data so that differences in feature ranges—heights within 50–100 inches or incomes from \$20,000 to \$200,000—don't incur undo weight penalties or benefits just from their scale. We'll discuss standardization and scaling more in Section 10.3. The categorical values in *diabetes* were recorded numerically as $\{0, 1\}$ and then standardized. I mention it to explain why there are *negative* ages (the mean age is zero after standardizing) and why the sexes are coded, or recorded, as $\{0.0507, -0.0446\}$ instead of $\{M, F\}$.

In [2]:

```
diabetes = datasets.load_diabetes()

tts = skms.train_test_split(diabetes.data,
                            diabetes.target,
                            test_size=.25)

(diabetes_train_ftrs, diabetes_test_ftrs,
 diabetes_train_tgt,  diabetes_test_tgt) = tts
```

We can dress the dataset up with a `DataFrame` and look at the first few rows:

In [3]:

```
diabetes_df = pd.DataFrame(diabetes.data,
                           columns=diabetes.feature_names)
diabetes_df['target'] = diabetes.target
diabetes_df.head()
```

Out[3]:

	age	sex	bmi	bp	s1	s2	s3	s4	s5	s6	target
0	0.04	0.05	0.06	0.02	-0.04	-0.03	-0.04	0.00	0.02	-0.02	151.00
1	0.00	-0.04	-0.05	-0.03	-0.01	-0.02	0.07	-0.04	-0.07	-0.10	75.00
2	0.09	0.05	0.04	-0.01	-0.05	-0.03	-0.03	0.00	0.00	-0.03	141.00
3	-0.09	-0.04	-0.01	-0.04	0.01	0.02	-0.04	0.03	0.02	-0.01	206.00
4	0.01	-0.04	-0.04	0.02	0.00	0.02	0.01	0.00	-0.03	-0.05	135.00

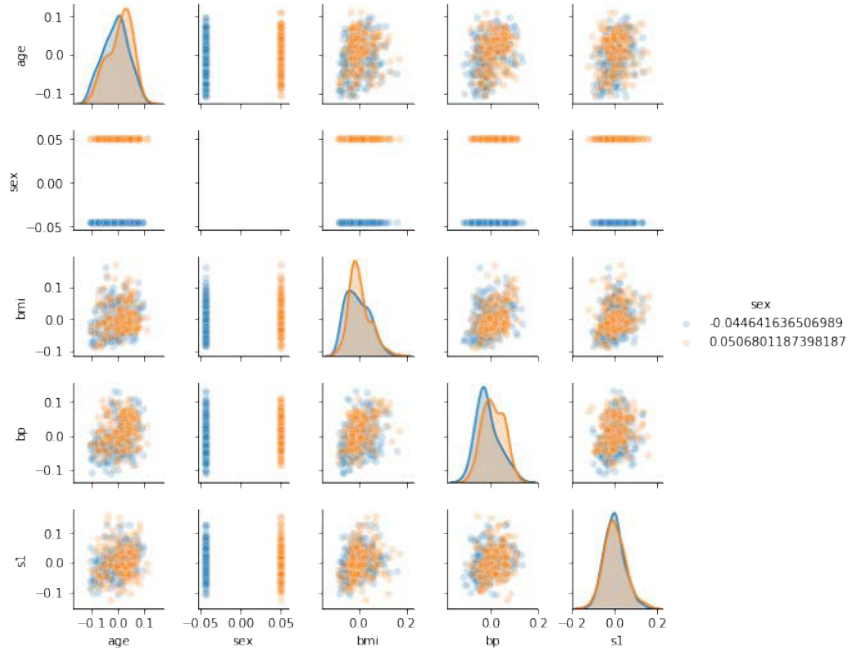
Aside from the odd values for seemingly categorical measures like age and sex, two of the other columns are quickly explainable; the rest are more specialized and somewhat underspecified:

- *bmi* is the *body mass index*, computed from height and weight, which is an approximation of body-fat percentage,
- *bp* is the *blood pressure*,
- *s1–s6* are six *blood serum measurements*, and
- *target* is a numerical score measuring the progression of a patient's illness.

As we did with the *iris* data, we can investigate the bivariate relationships with Seaborn's `pairplot`. We'll keep just a subset of the measurements for this graphic. The resulting mini-plots are still fairly small, but we can still glance through them and look for overall patterns. We can always redo the `pairplot` with all the features if we want to zoom out for a more global view.

In [4]:

```
sns.pairplot(diabetes_df[['age', 'sex', 'bmi', 'bp', 's1']],
             size=1.5, hue='sex', plot_kws={'alpha':.2});
```



4.2 Nearest-Neighbors Regression and Summary Statistics

We discussed nearest-neighbor classification in the previous chapter and we came up with the following sequence of steps:

1. Describe similarity between pairs of examples.
2. Pick several of the most-similar examples.
3. Combine the picked examples into a single answer.

As we shift our focus from predicting a class or category to predicting a numerical value, steps 1 and 2 can stay the same. Everything that we said about them still applies. However, when we get to step 3, we have to make adjustments. Instead of simply voting for candidate answers, we now need to take into account the quantities represented by the outputs. To do this, we need to combine the numerical values into a single, representative answer. Fortunately, there are several handy techniques for calculating a single summary value from a set of values. Values computed from a set of data are called *statistics*. If we are trying to represent—to summarize, if you will—the overall dataset with one of these, we call it a *summary statistic*. Let's turn our attention to two of these: the median and the mean.

4.2.1 Measures of Center: Median and Mean

You may be familiar with the average, also called the arithmetic mean. But I'm going start with a—seemingly!—less math-heavy alternative: the median. The median of a group of numbers is the *middle* number when that group is written in order. For example, if I have three numbers, listed in order as $[1, 8, 10]$, then 8 is the middle value: there is one number above it and one below. Within a group of numbers, the median has the same count of values below it and above it. To put it another way, if all the numbers have equal weight, regardless of their numerical value, then a scale placed at the median would be balanced (Figure 4.1). Regardless of the biggest value on the right—be it 15 or 40—the median stays the same.

You might be wondering what to do when we have an *even* number of values, say $[1, 2, 3, 4]$. The usual way to construct the median is to take the middle *two* values—the 2 and 3—and take their average, which gives us 2.5. Note that there are still the same number of values, two, above and below this median.

Using the median as a summary statistic has one wonderful property. If I fiddle with the values of the numbers at the start or end of the sorted data, the median stays the same. For example, if my data recorder is fuzzy towards the tails—i.e., values far from the median—and instead of $[1, 8, 10]$, I record $[2, 8, 11]$, my median is the same! This resilience in the face of differing measured values is called *robustness*. The median is a *robust* measure of center.

Now, there are scenarios where we care about the actual numbers, not just their in-order positions. The other familiar way of estimating the center is the *mean*. Whereas the median balances the *count* of values to the left and right, the mean balances the total distances to the left and right. So, the mean is the value for which `sum(distance(s,mean) for s in smaller)` is equal to `sum(distance(b,mean) for b in bigger)`. The only value that meets this constraint is `mean=sum(d)/len(d)` or, as the mathematicians say, mean $= \bar{x} = \frac{\sum_i x_i}{n}$. Referring back to Figure 4.1, if we trade the 15 for a 40, we get a different balance point: the mean has increased because the sum of the values has increased.

The benefit of the mean is that it accounts for the specific numeric values of the numbers: the value 3 is five units below the mean of 8. Compare to the median which

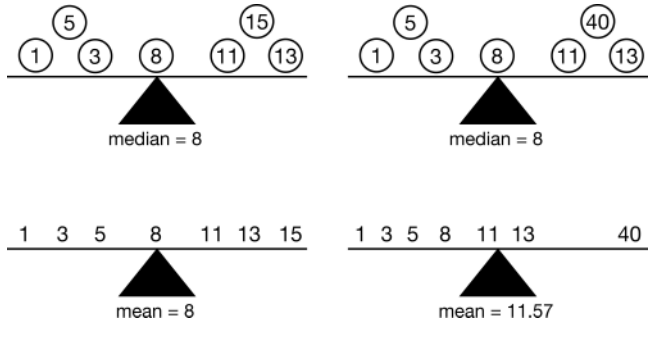


Figure 4.1 Comparing mean and median with balances.

abstracts distance away in favor of ordering: the value 3 is less than the median 8. The problem with the mean is that if we get an outlier—a rare event near the tails of our data—it can badly skew our computation precisely because the specific value matters.

As an example, here's what happens if we shift one value by "a lot" and recompute the mean and median:

In [5]:

```
values = np.array([1, 3, 5, 8, 11, 13, 15])
print("no outlier")
print(np.mean(values),
      np.median(values))

values_with_outlier = np.array([1, 3, 5, 8, 11, 13, 40])
print("with outlier")
print("%5.2f" % np.mean(values_with_outlier),
      np.median(values_with_outlier))
```

```
no outlier
8.0 8.0
with outlier
11.57 8.0
```

Beyond the mean and median, there are many possible ways to combine the nearest-neighbor answers into an answer for a test example. One combiner that builds on the idea of the mean is a *weighted* mean which we discussed in Section 2.5.1. In the nearest-neighbor context, we have a perfect candidate to serve as the weighting factor: the distance from our new example to the neighbor. So, instead of neighbors contributing just their values $[4.0, 6.0, 8.0]$, we can also incorporate the distance from each neighbor to our example. Let's say those distances are $[2.0, 4.0, 4.0]$, i.e. the second and third training examples are twice as far from our test example as the first one. A simple way to incorporate the distance is to compute a *weighted average* using

In [6]:

```
distances = np.array([2.0, 4.0, 4.0])
closeness = 1.0 / distances             # element-by-element division
weights = closeness / np.sum(closeness)  # normalize sum to one
weights
```

Out[6]:

```
array([0.4, 0.2, 0.2])
```

Or, in mathese:

$$\frac{\frac{1}{\text{distances}}}{\sum\left(\frac{1}{\text{distances}}\right)}$$

as the weights. We use $\frac{1}{\text{distances}}$ since if you are *closer*, we want a *higher* weight; if you are *further*, but still a nearest neighbor, we want a *lower* weight. We put the entire sum into the numerator to normalize the values so they sum to one. Compare the mean with the weighted mean for these values:

In [7]:

```
values = np.array([4, 6, 8])

mean = np.mean(values)
wgt_mean = np.dot(values, weights)

print("Mean:", mean)
print("Weighted Mean:", wgt_mean)
```

```
Mean: 6.0
Weighted Mean: 6.4
```

Graphically—see Figure 4.2—our balance diagram now looks a bit different. The examples that are downweighted (contribute less than their fair share) move closer to the pivot because they have less mechanical leverage. Overweighted examples move away from the pivot and gain more influence.

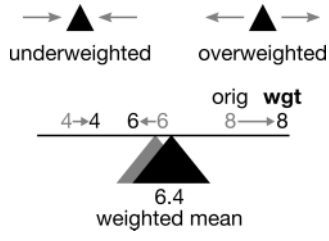


Figure 4.2 The effects of weighting on a mean.

4.2.2 Building a *k*-NN Regression Model

Now that we have some mental concepts to back up our understanding of *k*-NN regression, we can return to our basic `sklearn` workflow: build, fit, predict, evaluate.

In [8]:

```
knn   = neighbors.KNeighborsRegressor(n_neighbors=3)
fit   = knn.fit(diabetes_train_ftrs, diabetes_train_tgt)
preds = fit.predict(diabetes_test_ftrs)

# evaluate our predictions against the held-back testing targets
metrics.mean_squared_error(diabetes_test_tgt, preds)
```

```
Out[8]:
3471.41941941942
```

If you flip back to the previous chapter and our *k*-NN classifier, you'll notice only two differences.

1. We built a different model: this time we used `KNeighborsRegressor` instead of `KNeighborsClassifier`.
2. We used a different evaluation metric: this time we used `mean_squared_error` instead of `accuracy_score`.

Both of these reflect the difference in the targets we are trying to predict—numerical values, not Boolean categories. I haven't explained `mean_squared_error` (MSE) yet; it's because it is deeply tied to our next learning method, linear regression, and once we understand linear regression, we'll basically understand MSE for free. So, just press pause on evaluating regressors with MSE for a few minutes. Still, if you need *something* to make you feel comfortable, take a quick look at Section 4.5.1.

To put the numerical value for MSE into context, let's look at two things. First, the MSE is approximately 3500. Let's take its square root—since we're adding up squares, we need to scale back to nonsquares:

```
In [9]:
np.sqrt(3500)
```

```
Out[9]:
59.16079783099616
```

Now, let's look at the range of values that the target can take:

```
In [10]:
diabetes_df['target'].max() - diabetes_df['target'].min()
```

```
Out[10]:
321.0
```

So, the target values span about 300 units and our predictions are off—in some average sense—by 60 units. That's around 20%. Whether or not that is "good enough" depends on many other factors which we'll see in Chapter 7.

4.3 Linear Regression and Errors

We're going to dive into linear regression (LR)—which is just a fancy name for drawing a straight line through a set of data points. I'm really sorry to bring up LR, particularly if you are one of those folks out there who had a bad experience studying LR previously. LR has a long history throughout math and science. You may have been exposed to it before. You may have seen LR in an algebra or statistics class. Here's a very different presentation.

4.3.1 No Flat Earth: Why We Need Slope

Do you like to draw? (If not, please just play along.) Take a pen and draw a bunch of dots on a piece of paper. Now, draw a single straight line through the dots. You might have encountered a problem already. If there were more than two dots, there are many, many different lines you could potentially draw. The idea of *drawing a line through the dots* gets a general idea across, but it doesn't give us a reproducible way of specifying or completing the task.

One way of picking a specific line through the dots is to say we want a *best* line—problem solved. We're done. Let's go for five o'clock happy hour. Oh, wait, I have to define what *best* means. Rats! Alright, let's try this. I want the line that stays closest to the dots based on the *vertical* distance from the dots to the line. Now we're getting somewhere. We have something we can calculate to compare different alternatives.

Which line is best under that criteria? Let me start by simplifying a little bit. Imagine we can only draw lines that are parallel to the bottom of the piece of paper. You can think about moving the line like raising or lowering an Olympic high-jump bar: it stays parallel to the ground. If I start sliding the line up and down, I'm going to start far away from all the points, move close to some points, slide onwards to a great, just-right middle ground, move close to other points, and finally end up far away from everything. Yes, the idea of a happy medium—too hot, too cold, and just right—applies here. We'll see this example in code and graphics in just a moment. At the risk of becoming too abstract too quickly, we are limited to drawing lines like $y = c$. In English, that means the height of our bar is always equal to some constant, fixed value.

Let's draw a few of these high-jump bars and see what happens.

In [11]:

```
def axis_helper(ax, lims):
    'clean up axes'
    ax.set_xlim(lims); ax.set_xticks([])
    ax.set_ylim(lims); ax.set_yticks([])
    ax.set_aspect('equal')
```

We're going to use some trivial data to show what's happening.

In [12]:

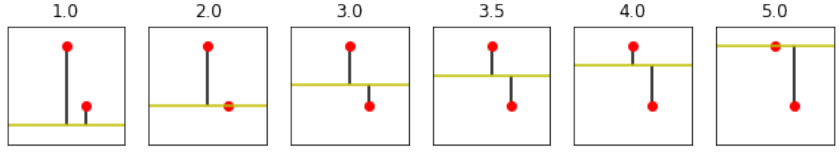
```
# our data is very simple:  two (x, y) points
D = np.array([[3, 5],
              [4, 2]])

# we'll take x as our "input" and y as our "output"
x, y = D[:, 0], D[:, 1]
```

Now, let's graph what happens as we move a horizontal line up through different possible values. We'll call these values our *predicted* values. You can imagine each raised bar as being a possible set of predictions for our example data points. Along the way, we'll also

keep track of what our error values are. The errors are the differences between the horizontal line and the data points. We'll also calculate a few values from the errors: the sum of errors, the *sum of squares* of the errors (abbreviated SSE), and the *square root of the sum of squared errors*. You might want to look at the output first, before trying to understand the code.

In [13]:

```
horizontal_lines = np.array([1, 2, 3, 3.5, 4, 5])

results = []
fig, axes = plt.subplots(1, 6, figsize=(10, 5))
for h_line, ax in zip(horizontal_lines, axes.flat):
    # styling
    axis_helper(ax, (0, 6))
    ax.set_title(str(h_line))

    # plot the data
    ax.plot(x, y, 'ro')

    # plot the prediction line
    ax.axhline(h_line, color='y') # ax coords; defaults to 100%

    # plot the errors
    # the horizontal line *is* our prediction; renaming for clarity
    predictions = h_line
    ax.vlines(x, predictions, y)

    # calculate the error amounts and their sum of squares
    errors = y - predictions
    sse = np.dot(errors, errors)

    # put together some results in a tuple
    results.append((predictions,
                    errors, errors.sum(),
                    sse, np.sqrt(sse)))
```



We start very far away from one point and not too far from another. As we slide the bar up, we hit a nice middle ground between the points. Yet we keep sliding the bar; we

end up on the top point, fairly far away from the bottom point. Perhaps the ideal tradeoff is somewhere in the middle. Let's look at some numbers.

In [14]:

```
col_labels = "Prediction", "Errors", "Sum", "SSE", "Distance"
display(pd.DataFrame.from_records(results,
                                  columns=col_labels,
                                  index="Prediction"))
```

Prediction	Errors	Sum	SSE	Distance
1.0000	[4.0, 1.0]	5.0000	17.0000	4.1231
2.0000	[3.0, 0.0]	3.0000	9.0000	3.0000
3.0000	[2.0, -1.0]	1.0000	5.0000	2.2361
3.5000	[1.5, -1.5]	0.0000	4.5000	2.1213
4.0000	[1.0, -2.0]	-1.0000	5.0000	2.2361
5.0000	[0.0, -3.0]	-3.0000	9.0000	3.0000

Our table includes the raw errors that can be positive or negative—we might over- or underestimate. The sums of those raw errors don't do a great job evaluating the lines. Errors in the opposite directions, such as $[2, -1]$, give us a total of 1. In terms of our overall prediction ability, we don't want these errors to cancel out. One of the best ways to address that is to use a *total distance*, just like we used distances in the nearest-neighbors method above. That means we want something like $\sqrt{(\text{prediction} - \text{actual})^2}$. The SSE column is the *sum of squared errors* which gets us most of the way towards calculating distances. All that's left is to take a square root. The line that is best, under the rules so far, is the horizontal line at the mean of the points based on their vertical component: $\frac{5+2}{2} = 3.5$. The mean is the best answer here for the same reason it is the pivot on the balance beams we showed earlier: it perfectly balances off the errors on either side.

4.3.2 Tilting the Field

What happens if we keep the restriction of drawing straight lines but remove the restriction of making them horizontal? My apologies for stating the possibly obvious, but now we can draw lines that aren't flat. We can draw lines that are pointed up or down, as needed. So, if the cluster of dots that you drew had an overall growth or descent, like a plane taking off or landing, a sloped line can do better than a flat, horizontal runway. The form of these sloped lines is a classic equation from algebra: $y = mx + b$. We get to adjust m and b so that when we know a point's left-rightedness—the x value—we can get as close as possible on its up-downness y.

How do we define close? The same way we did above—with distance. Here, however, we have a more interesting line with a slope. What does the distance from a point to our

line look like? It's just `distance(prediction, y) = distance(m*x + b,y)`. And what's our total distance? Just add those up for all of our data: `sum(distance(mx + b, y) for x, y in D)`. In mathese, that looks like

$$\sum_{x,y \in D} \big((mx + b) - y\big)^2$$

I promise code and graphics are on the way! A side note: it's possible that for a set of dots, the best line *is* flat. That would mean we want an answer that is a simple, horizontal line—just what we discussed in the previous section. When that happens, we just set m to zero and head on our merry way. Nothing to see here, folks. Move along.

Now, let's repeat the horizontal experiment with a few, select tilted lines. To break things up a bit, I've factored out the code that draws the graphs and calculates the table entries into a function called `process`. I'll admit `process` is a *horrible* name for a function. It's up there with *stuff* and *things*. Here, though, consider it the processing we do with our small dataset and a simple line.

In [15]:

```
def process(D, model, ax):
    # make some useful abbreviations/names
    # y is our "actual"
    x, y = D[:, 0], D[:, 1]
    m, b = model

    # styling
    axis_helper(ax, (0, 8))

    # plot the data
    ax.plot(x, y, 'ro')

    # plot the prediction line
    helper_xs = np.array([0, 8])
    helper_line = m * helper_xs + b
    ax.plot(helper_xs, helper_line, color='y')

    # plot the errors
    predictions = m * x + b
    ax.vlines(x, predictions, y)

    # calculate error amounts
    errors = y - predictions

    # tuple up the results
```

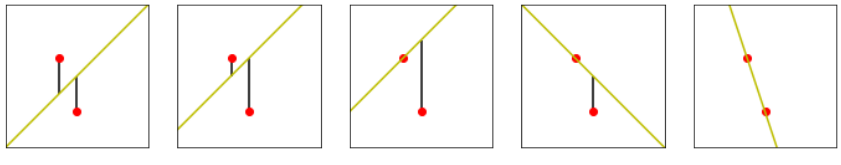
```
    sse = np.dot(errors, errors)
    return (errors, errors.sum(), sse, np.sqrt(sse))
```

Now we'll make use of `process` with several different prediction lines:

In [16]:

```
# our data is very simple:  two (x, y) points
D = np.array([[3, 5],
              [4, 2]])

#                      m   b  --> predictions = mx + b
lines_mb = np.array([[ 1,  0],
                     [ 1,  1],
                     [ 1,  2],
                     [-1,  8],
                     [-3, 14]])

col_labels = ("Raw Errors", "Sum", "SSE", "TotDist")
results = []

# note: plotting occurs in process()
fig, axes = plt.subplots(1, 5, figsize=(12, 6))
records = [process(D, mod, ax) for mod,ax in zip(lines_mb, axes.flat)]
df = pd.DataFrame.from_records(records, columns=col_labels)
display(df)
```

	Raw Errors	Sum	SSE	TotDist
0	[2, -2]	0	8	2.8284
1	[1, -3]	-2	10	3.1623
2	[0, -4]	-4	16	4.0000
3	[0, -2]	-2	4	2.0000
4	[0, 0]	0	0	0.0000



So, we have a progression in calculating our measure of success:

- `predicted = m*x + b`
- `error = (m*x + b) - actual = predicted - actual`
- `SSE = sum(errors**2) = sum(((m*x+b) - actual)**2 for x,actual in data)`
- `total_distance = sqrt(SSE)`

The last line precisely intersects with both data points. Its predictions are 100% correct; the vertical distances are zero.

4.3.3 Performing Linear Regression

So far, we've only considered what happens with a single predictive feature `x`. What happens when we add more features—more columns and dimensions—into our model? Instead of a single slope `m`, we now have to deal with a slope for each of our features. We have some contribution to the outcome from each of the input features. Just as we learned how our output changes with one feature, we now have to account for different relative contributions from different features.

Since we have to track many different slopes—one for each feature—we're going to shift away from using `m` and use the term *weights* to describe the contribution of each feature. Now, we can create a linear combination—as we did in Section 2.5—of the weights and the features to get the prediction for an example. The punchline is that our prediction is `rdot(weights_wo, features) + wgt_b` if `weights_wo` is *without* the `b` part included. If we use the plus-one trick, it is `rdot(weights, features_p1)` where `weights` includes a `b` (as `weights[0]`) and `features_p1` includes a column of ones. Our error still looks like `distance(prediction, actual)` with `prediction=rdot(weights, features_p1)`. The mathese form of a prediction (with a prominent dot product) looks like:

$$y_{\text{pred}} = \sum_{\text{ftrs}} w_f x_f = w \bullet x$$

In [17]:

```
lr    = linear_model.LinearRegression()
fit   = lr.fit(diabetes_train_ftrs, diabetes_train_tgt)
preds = fit.predict(diabetes_test_ftrs)

# evaluate our predictions against the unseen testing targets
metrics.mean_squared_error(diabetes_test_tgt, preds)
```

Out[17]:

```
2848.2953079329427
```

We'll come back to `mean_squared_error` in just a minute, but you are already equipped to understand it. It is the average distance of the errors in our prediction.

4.4 Optimization: Picking the Best Answer

Picking the best line means picking the best values for m and b or for the *weights*. In turn, that means setting factory knobs to their best values. How can we choose these *bests* in a well-defined way?

Here are four strategies we can adopt:

1. Random guess: Try lots of possibilities at random, take the best one.
2. Random step: Try one line—pick an m and a b—at random, make several random adjustments, pick the adjustment that helps the most. Repeat.
3. Smart step: Try one line at random, see how it does, adjust it in some smart way. Repeat.
4. Calculated shortcut: Use fancy mathematics to prove that if Fact A, Fact B, and Fact C are all true, then the One Line To Rule Them All must be the best. Plug in some numbers and use The One Line To Rule Them All.

Let's run through these using a really, really simple constant-only model. Why a constant, you might ask. Two reasons. First, it is a simple horizontal line. After we calculate its value, it is the same everywhere. Second, it is a simple baseline for comparison. If we do well with a simple constant predictor, we can just call it a day and go home. On the other hand, if a more complicated model does as well as a simple constant, we might question the value of the more complicated model. As Yoda might say, "A simple model, never underestimate."

4.4.1 Random Guess

Let's make some simple data to predict.

In [18]:

```
tgt = np.array([3, 5, 8, 10, 12, 15])
```

Let's turn Method 1—random guessing—into some code.

In [19]:

```
# random guesses with some constraints
num_guesses = 10
results = []

for g in range(num_guesses):
    guess = np.random.uniform(low=tgt.min(), high=tgt.max())
    total_dist = np.sum((tgt - guess)**2)
    results.append((total_dist, guess))
best_guess = sorted(results)[0][1]
best_guess
```

Out[19]:

```
8.228074784134693
```

Don't read too much into this specific answer. Just keep in mind that, since we have a simple value to estimate, we only need to take a few shots to get a good answer.

4.4.2 Random Step

Method 2 starts with a single random guess, but then takes a random step up or down. If that step is an improvement, we keep it. Otherwise, we go back to where we were.

In [20]:

```
# use a random choice to take a hypothetical
# step up or down:  follow it, if it is an improvement
num_steps = 100
step_size = .05

best_guess = np.random.uniform(low=tgt.min(), high=tgt.max())
best_dist  = np.sum((tgt - best_guess)**2)

for s in range(num_steps):
    new_guess = best_guess + (np.random.choice([+1, -1]) * step_size)
    new_dist = np.sum((tgt - new_guess)**2)
    if new_dist < best_dist:
        best_guess, best_dist = new_guess, new_dist
print(best_guess)
```

```
8.836959712695537
```

We start with a single guess and then try to improve it by random stepping. If we take enough steps and those steps are individually small enough, we should be able to find our way to a solid answer.

4.4.3 Smart Step

Imagine walking, blindfolded, through a rock-strewn field or a child's room. You might take tentative, test steps as you try to move around. After a step, you use your foot to probe the area around you for a clear spot. When you find a clear spot, you take that step.

In [21]:

```
# hypothetically take both steps (up and down)
# choose the better of the two
# if it is an improvement, follow that step
```

```
num_steps = 1000
step_size = .02

best_guess = np.random.uniform(low=tgt.min(), high=tgt.max())
best_dist  = np.sum((tgt - best_guess)**2)
print("start:", best_guess)
for s in range(num_steps):
    # np.newaxis is needed to align the minus
    guesses = best_guess + (np.array([-1, 1]) * step_size)
    dists   = np.sum((tgt[:,np.newaxis] - guesses)**2, axis=0)

    better_idx = np.argmin(dists)

    if dists[better_idx] > best_dist:
        break

    best_guess = guesses[better_idx]
    best_dist  = dists[better_idx]
print("  end:", best_guess)
```

```
start: 9.575662598977047
  end: 8.835662598977063
```

Now, unless we get stuck in a bad spot, we should have a better shot at success than random stepping: at any given point we check out the legal alternatives and take the best of them. By effectively cutting out the random steps that don't help, we should make progress towards a good answer.

4.4.4 Calculated Shortcuts

If you go to a statistics textbook, you'll discover that for our SSE evaluation criteria, there is a *formula* for the answer. To get the smallest sum of squared errors, what we need is precisely the *mean*. When we said earlier that the mean balanced the distances to the values, we were merely saying the same thing in a different way. So, we don't actually have to search to find our best value. The fancy footwork is in the mathematics that demonstrates that the mean is the right answer to this question.

In [22]:

```
print("mean:", np.mean(tgt))
```

```
mean: 8.833333333333334
```

4.4.5 Application to Linear Regression

We can apply these same ideas to fitting a sloped line, or finding many weights (one per feature), to our data points. The model becomes a bit more complicated—we have to twiddle more values, either simultaneously or in sequence. Still, it turns out that an equivalent to our Method 4, *Calculated Shortcut*, is the standard, classical way to find the best line. When we fit a line, the process is called *least-squares fitting* and it is solved by the *normal equations*—you don't have to remember that—instead of just the *mean*. Our Method 3, *Smart Step*, using some mathematics to limit the direction of our steps, is common when dealing with very big data where we can't run all the calculations needed for the standard method. That method is called *gradient descent*. Gradient descent (GD) uses some smart calculations—instead of probing steps—to determine directions of improvement.

The other two methods are not generally used to find a best line for linear regression. However, with some additional details, Method 2, *Random Step*, is close to the techniques of *genetic algorithms*. What about Method 1, *Random Guessing*? Well, it isn't very useful by itself. But the idea of random starting points *is* useful when combined with other methods. This discussion is just a quick introduction to these ideas. We'll mention them throughout the book and play with them in Chapter 15.

4.5 Simple Evaluation and Comparison of Regressors

Earlier, I promised we'd come back to the idea of mean squared error (MSE). Now that we've discussed sum of squared errors and total distances from a regression line, we can tie these ideas together nicely.

4.5.1 Root Mean Squared Error

How can we quantify the performance of regression predictions? We're going to use some mathematics that are almost identical to our criteria for finding good lines. Basically, we'll take the average of the squared errors. Remember, we can't just add up the errors themselves because then a $+3$ and a -3 would cancel each other out and we'd consider those predictions perfect when we're really off by a total of 6. Squaring and adding those two values gives us a total error of 18. Averaging gives us a mean squared error of 9. We'll take one other step and take the *square root* of this value to get us back to the same scale as the errors themselves. This gives us the *root mean squared error*, often abbreviated *RMSE*. Notice that in this example, our RMSE is 3: precisely the amount of the error(s) in our individual predictions.

That reminds me of an old joke for which I can't find specific attribution:

> Two statisticians are out hunting when one of them sees a duck. The first takes aim and shoots, but the bullet goes sailing past six inches too high. The second statistician also takes aim and shoots, but this time the bullet goes sailing past six inches too low. The two statisticians then give one another high fives and exclaim, "Got him!"

Groan all you want, but that is the fundamental tradeoff we make when we deal with averages. Please note, no ducks were harmed in the making of this book.

4.5.2 Learning Performance

With data, methods, and an evaluation metric in hand, we can do a small comparison between *k*-NN-R and LR.

In [23]:

```
# stand-alone code
from sklearn import (datasets, neighbors,
                     model_selection as skms,
                     linear_model, metrics)

diabetes = datasets.load_diabetes()
tts =  skms.train_test_split(diabetes.data,
                             diabetes.target,
                             test_size=.25)
(diabetes_train, diabetes_test,
 diabetes_train_tgt, diabetes_test_tgt) = tts

models = {'kNN': neighbors.KNeighborsRegressor(n_neighbors=3),
          'linreg' : linear_model.LinearRegression()}

for name, model in models.items():
    fit   = model.fit(diabetes_train, diabetes_train_tgt)
    preds = fit.predict(diabetes_test)

    score = np.sqrt(metrics.mean_squared_error(diabetes_test_tgt, preds))
    print("{:>6s} : {:0.2f}".format(name,score))
```

```
   kNN : 54.85
linreg : 46.95
```

4.5.3 Resource Utilization in Regression

Following Section 3.7.3, I wrote some stand-alone test scripts to get an insight into the resource utilization of these regression methods. If you compare the code here with the earlier code, you'll find only two differences: (1) different learning methods and (2) a different learning performance metric. Here is that script adapted for *k*-NN-R and LR:

In [24]:

```
!cat scripts/perf_02.py
```

```
import timeit, sys
import functools as ft
import memory_profiler
from mlwpy import *

def knn_go(train_ftrs, test_ftrs, train_tgt):
    knn = neighbors.KNeighborsRegressor(n_neighbors=3)
    fit   = knn.fit(train_ftrs, train_tgt)
    preds = fit.predict(test_ftrs)

def lr_go(train_ftrs, test_ftrs, train_tgt):
    linreg = linear_model.LinearRegression()
    fit   = linreg.fit(train_ftrs, train_tgt)
    preds = fit.predict(test_ftrs)

def split_data(dataset):
    split = skms.train_test_split(dataset.data,
                                  dataset.target,
                                  test_size=.25)
    return split[:-1] # don't need test tgt

def msr_time(go, args):
    call = ft.partial(go, *args)
    tu = min(timeit.Timer(call).repeat(repeat=3, number=100))
    print("{:<6}: ~{:.4f} sec".format(go.__name__, tu))

def msr_mem(go, args):
    base = memory_profiler.memory_usage()[0]
    mu = memory_profiler.memory_usage((go, args),
                                      max_usage=True)[0]
    print("{:<3}: ~{:.4f} MiB".format(go.__name__, mu-base))

if __name__ == "__main__":
    which_msr = sys.argv[1]
    which_go = sys.argv[2]

    msr = {'time': msr_time, 'mem':msr_mem}[which_msr]
    go = {'lr' : lr_go, 'knn': knn_go}[which_go]

    sd = split_data(datasets.load_iris())
    msr(go, sd)
```

When we execute it, we see

```
In [25]:
```

```
!python scripts/perf_02.py mem lr
!python scripts/perf_02.py time lr
```

```
lr_go: ~1.5586 MiB
lr_go : ~0.0546 sec
```

```
In [26]:
```

```
!python scripts/perf_02.py mem knn
!python scripts/perf_02.py time knn
```

```
knn_go: ~0.3242 MiB
knn_go: ~0.0824 sec
```

Here's a brief table of our results that might vary a bit over different runs:

Method	RMSE	Time (s)	Memory (MiB)
k-NN-R	55	0.08	0.32
LR	45	0.05	1.55

It may be surprising that linear regression takes up so much memory, especially considering that *k*-NN-R requires keeping all the data around. This surprise highlights an issue with the way we are measuring memory: (1) we are measuring the *entire* fit-and-predict process as one unified task and (2) we are measuring the *peak* usage of that unified task. Even if linear regression has one brief moment of high usage, that's what we are going to see. Under the hood, this form of linear regression—which optimizes by Method 4, *Calculated Shortcut*—isn't super clever about how it does its calculations. There's a critical part of its operation—solving those normal equations I mentioned above—that is very memory hungry.

4.6 EOC

4.6.1 Limitations and Open Issues

There are several caveats to what we've done in this chapter—and many of them are the same as the previous chapter:

- We compared these learners on a single dataset.
- We used a very simple dataset.
- We did *no* preprocessing on the dataset.
- We used a single train-test split.

- We used accuracy to evaluate the performance.
- We didn't try different numbers of neighbors.
- We only compared two simple models.

Additionally, linear regression is quite sensitive to using standardized data. While *diabetes* came to us prestandardized, we need to keep in mind that *we* might be responsible for that step in other learning problems. Another issue is that we can often benefit from restricting the weights—the $\{m, b\}$ or w—that a linear regression model can use. We'll talk about why that is the case and how we can do it with `sklearn` in Section 9.1.

4.6.2 Summary

Wrapping up our discussion, we've seen several things in this chapter:

1. *diabetes*: a simple real-world dataset
2. Linear regression and an adaptation of nearest-neighbors for regression
3. Different measures of center—the mean and median
4. Measuring learning performance with root mean squared error (RMSE)

4.6.3 Notes

The *diabetes* data is from a paper by several prominent statisticians which you can read here: http://statweb.stanford.edu/~tibs/ftp/lars.pdf.

4.6.4 Exercises

You might be interested in trying some classification problems on your own. You can follow the model of the sample code in this chapter with another regression dataset from `sklearn`: `datasets.load_boston` will get you started!

This page intentionally left blank

Part II

Evaluation

This page intentionally left blank

5

Evaluating and Comparing Learners

In [1]:

```
# setup
from mlwpy import *
diabetes = datasets.load_diabetes()
%matplotlib inline
```

5.1 Evaluation and Why Less Is More

> *Lao Tzu:* Those that know others are wise. Those that know themselves are Enlightened.

The biggest risk in developing a learning system is *overestimating how well it will do when we use it*. I touched on this risk in our first look at classification. Those of us that have studied for a test and *thought* we had a good mastery of the material, and then *bombed* the test, will be intimately familiar with this risk. It is very, very easy to (1) think we know a lot and will do well on an exam and (2) not do very well on the exam. On a test, we may discover we need details when we only remember a general idea. I know it happened in mid-nineteenth century, but was it 1861 or 1862!? Even worse, we might focus on some material at the expense of other material: we might miss studying some information entirely. Well, *nuts*: we needed to know her name but not his birth year.

In learning systems, we have two similar issues. When you study for the test, you are *limited* in what you can remember. Simply put, your brain gets full. You don't have the *capacity* to learn each and every detail. One way around this is to remember the big picture instead of many small details. It is a great strategy—until you need one of those details! Another pain many of us have experienced is that when you're studying for a test, your friend, spouse, child, *anyone* hollers at you, "I need your attention now!" Or it might be a new video game that comes out: "Oh look, a shiny bauble!" Put simply, you get *distracted by noise*. No one is judging, we're all human here.

These two pitfalls—limited capacity and distraction by noise—are shared by computer learning systems. Now, typically, a learning system won't be distracted by the latest YouTube sensation or Facebook meme. In the learning world, we call these sources of error by different names. For the impatient, they are *bias* for the capacity of what we can squeeze into our head and *variance* for how distracted we get by noise. For now, squirrel away that bit of intuition and don't get distracted by noise.

Returning to the issue of overconfidence, what can we do to protect ourselves from . . . ourselves? Our most fundamental defense is *not teaching to the test*. We introduced this idea in our first look at classification (Section 3.3). To avoid teaching to the test, we use a very practical three-step recipe:

- Step one: split our data into separate training and testing datasets.
- Step two: *learn* on the training data.
- Step three: *evaluate* on the testing data.

Not using *all* the data to *learn* may seem counterintuitive. Some folks—certainly none of *my* readers—could argue, "Wouldn't building a model on more data lead to better results?" Our humble skeptic has a good point. Using more data *should* lead to better estimates by our learner. The learner should have better parameters—better knob settings on our factory machine. However, there's a really big consequence of using *all* of the data for learning. *How would we know that a more-data model is better than a less-data model?* We have to *evaluate* both models somehow. If we teach to the test by learning and evaluating on *all* of the data, we are likely to overestimate our ability once we take our system into the big, scary, complex real world. The scenario is similar to studying a specific test from last year's class—wow, multiple choice, easy!—and then being tested on this year's exam which is *all essays*. Is there a doctor in the house? A student just passed out.

In this chapter, we will dive into general evaluation techniques that apply to both regression and classification. Some of these techniques will help us avoid teaching to the test. Others will give us ways of comparing and contrasting learners in very broad terms.

5.2 Terminology for Learning Phases

We need to spend a few minutes introducing some vocabulary. We need to distinguish between a few different phases in the machine learning process. We've hit on *training* and *testing* earlier. I want to introduce another phase called *validation*. Due to some historical twists and turns, I need to lay out clearly what I mean by these three terms—training, validation, and testing. Folks in different disciplines can use these terms with slight variations in meaning which can trip the unwary student. I want you to have a clear walking path.

5.2.1 Back to the Machines

I want you to return to the mental image of our factory learning machine from Section 1.3. The machine is a big black box of knobs, inputs, and outputs. I introduced that machine to give you a concrete image of what learning algorithms are doing and how

we have control over them. We can continue the story. While the machine itself seems to be *part* of a factory, in reality, we are a business-to-business (that's B2B to you early 2000s business students) provider. Other companies want to make use of our machine. However, they want a completely hands-off solution. We'll build the machine, set all the knobs as in Figure 5.1, and send the machine to the customer. They won't do anything other than feed it inputs and see what pops out the other side. This delivery model means that when we hand off the machine to our customer, it needs to be fully tuned and ready to rock-and-roll. Our challenge is to ensure the machine can perform adequately after the hand-off.

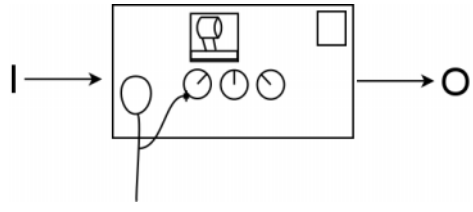


Figure 5.1 Learning algorithms literally dial-in—or optimize—a relationship between input and output.

In our prior discussion of the machine, we talked about relating inputs to outputs by setting the knobs and switches on the side of the machine. We established that relationship because we had some *known* outputs that we were expecting. Now, we want to avoid teaching to the test when we set the dials on our machine. We want the machine to do well for us, but more importantly, we want it to do well for our customer. Our strategy is to hold out some of the input-output pairs and save them for later. We will *not* use the saved data to set the knobs. We will use the saved data, *after* learning, to evaluate how well the knobs are set. Great! Now we're completely set and have a good process for making machines for our customers.

You know what's coming. Wait for it. Here it comes. Houston, we have a problem. There are many different types of machines that relate inputs to outputs. We've already seen two classifiers and two regressors. Our customers might have some preconceived ideas about what sort of machine they want because they heard that Fancy Silicon Valley Technologies, Inc. was using one type of machine. FSVT, Inc. might leave it entirely up to us to pick the machine. Sometimes we—or our corporate overlords—will choose between different machines based on characteristics of the inputs and outputs. Sometimes we'll choose based on resource use. Once we select a broad class of machines (for example, we decide we need a widget maker), there may be several physical machines we can pick (for example, the Widget Works 5000 or the WidgyWidgets Deluxe Model W would both work nicely). Often, we will pick the machine we use based on its learning performance (Figure 5.2).

Let me step out of the metaphor for a moment. A concrete example of a factory machine is a *k*-Nearest Neighbors (*k*-NN) classifier. For *k*-NN, different values of *k* are entirely different physical machines. *k* is *not* a knob we adjust on the machine. *k* is *internal*

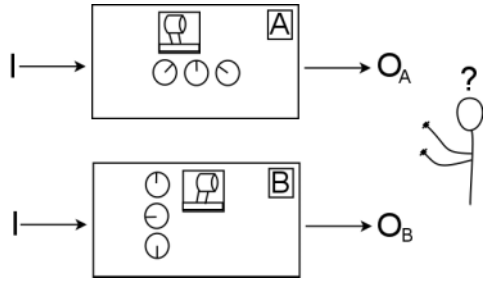


Figure 5.2 If we can build and optimize different machines, we select one of them for the customer.

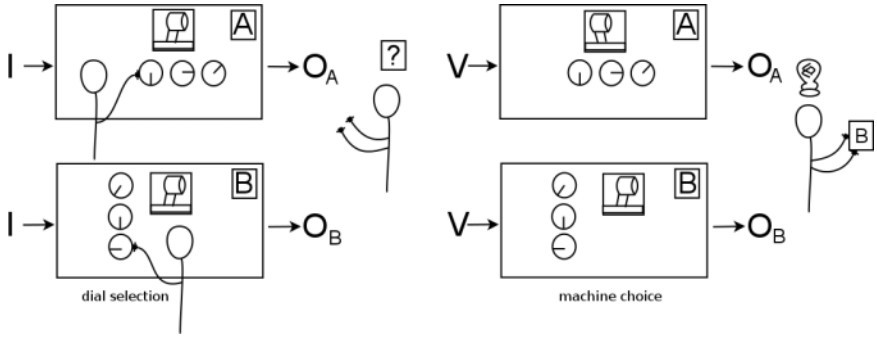


Figure 5.3 Optimization (dial-setting) and selection (machine choice) as steps to create a great machine for our customer.

to the machine. No matter what inputs and outputs we see, we can't adjust k directly on one machine (see Section 11.1 for details). It's like looking at the transmission of a car and wanting a different gearing. That modification is at a level beyond the skillsets of most of us. But all is not lost! We can't modify our car's transmission, but we *can* buy a different car. We are free to have *two different machines*, say 3-NN and 10-NN. We can go further. The machines could also be completely different. We could get two sedans and one minivan. With learning models, they don't all have to be k-NN variants. We could get a 3-NN, a 10-NN, and a Naive Bayes. To pick among them, we run our input-output pairs through the models to train them. Then, we evaluate how they perform on the held-out data to get a better—less trained on the test—idea of how our machines will perform for the customer (Figure 5.3).

Hurray! We're done. High-fives all around, it's time for coffee, tea, soda, or beer (depending on your age and doctor's advice).

Not so fast. We still have a problem. Just as we can teach to the test in setting the knobs on the machines, we can also teach to the test in terms of *picking a machine*. Imagine that

we use our held-out data as the basis for picking the best machine. For *k*-NN that means picking the best *k*. We could potentially try all the values of *k* up to the size of our dataset. Assuming 50 examples, that's all values of *k* from 1 to 50. Suppose we find that 27 is the best. That's great, except we've been looking at the same held-out data every time we try a different *k*. We no longer have an *unseen* test to give us a fair evaluation of the machine we're going to hand off to our customer. We used up our hold-out test set and fine-tuned our performance towards it. What's the answer now?

The answer to teaching-to-the-test with knobs—tuning a given machine—was to have a separate set of held-out data that isn't used to set the knobs. Since that worked pretty well there, let's just do that again. We'll have *two* sets of held-out data to deal with two separate steps. One set will be used to *pick the machine*. The second set will be used to *evaluate how well the machine will work for the customer* in a fair manner, without peeking at the test. Remember, we also have the non-held-out data that is used to tune the machine.

5.2.2 More Technically Speaking . . .

Let's recap. We now have three distinct sets of data. We can break the discussion of our needs into three distinct phases. We'll work from the outside to the inside—that is, from our final goal towards fitting the basic models.

1. We need to provide a single, well-tuned machine to our customer. We want to have a final, no-peeking evaluation of how that machine will do for our customer.
2. After applying some thought to the problem, we select a few candidate machines. With those candidate machines, we want to evaluate and compare them without peeking at the data we will use for our final evaluation.
3. For each of the candidate machines, we need to set the knobs to their best possible settings. We want to do that without peeking at either of the datasets used for the other phases. Once we select one machine, we are back to the basic learning step: we need to set its knobs.

5.2.2.1 Learning Phases and Training Sets

Each of these three phases has a component of evaluation in it. In turn, each different evaluation makes use of a specific set of data containing different known input-output pairs. Let's give the phases and the datasets some useful names. Remember, the term *model* stands for our metaphorical factory machine. The phases are

1. *Assessment*: final, last-chance estimate of how the machine will do when operating in the wild
2. *Selection*: evaluating and comparing different machines which may represent the same broad type of machine (different *k* in *k*-NN) or completely different machines (*k*-NN and Naive Bayes)
3. *Training*: setting knobs to their optimal values and providing auxiliary side-tray information

The datasets used for these phases are:

1. Hold-out test set
2. Validation test set
3. Training set

We can relate these phases and datasets to the factory machine scenario. This time, I'll work from the inside out.

1. The training set is used to adjust the knobs on the factory machine.
2. The validation test set is used to get a non-taught-to-the-test evaluation of that finely optimized machine and help us pick between different optimized machines.
3. The hold-out test set is used to make sure that *the entire process of building one or more factory machines, optimizing them, evaluating them, and picking among them* is evaluated fairly.

The last of these is a *big* responsibility: there are many ways to peek and be misled by distractions. If we train and validation-test over and over, we are building up a strong idea of what works and doesn't work in the validation test set. It may be indirect, but we are effectively *peeking* at the validation test set. The hold-out test set—data we have never used before in any training or validation-testing for this problem—is necessary to protect us from this *indirect peeking* and to give us a fair evaluation of how our final system will do with novel data.

5.2.2.2 Terms for Test Sets

If you check out a number of books on machine learning, you'll find the term *validation set* used, fairly consistently, for *Selection*. However, when you *talk* to practitioners, folks will verbally use the phrase *test set* for both datasets used for *Selection* and for *Assessment*. To sweep this issue under the carpet, if I'm talking about evaluation and it is either (1) clear from context or (2) doesn't particularly matter, I'll use the generic phrase *testing* for the data used in either *Assessment* or *Selection*. The most likely time that will happen is when we aren't doing *Selection* of models—we are simply using a basic train-test split, training a model and then performing a held-out evaluation, *Assessment*, on it.

If the terms *do* matter, as when we're talking about both phases together, I'll be a bit more precise and use more specific terms. If we need to distinguish these datasets, I'll use the terms *hold-out test set* (HOT) and *validation set* (ValS). Since *Assessment* is a one-and-done process—often at the end of all our hard work applying what we know about machine learning—we'll be talking about the HOT relatively infrequently. That is not to say that the HOT is unimportant—quite the contrary. Once we use it, we can never use it as a HOT again. We've peeked. Strictly speaking, we've contaminated both ourselves and our learning system. We can delete a learning system and start from scratch, but it is very difficult to erase our own memories. If we do this repeatedly, we'd be right back into teaching to the test. The only solution for breaking the lockbox of the HOT is

to gather new data. On the other hand, we are not obligated to use all of the HOT at once. We can use half of it, find we don't like the results and go back to square one. When we develop a new system that we need to evaluate before deployment, we still have the other half the HOT for *Assessment*.

5.2.2.3 A Note on Dataset Sizes

A distinctly practical matter is figuring out how big each of these sets should be. It is a difficult question to answer. If we have *lots* of data, then all three sets can be very large and there's no issue. If we have very little data, we have to be concerned with (1) using enough data in training to build a good model and (2) leaving enough data for the testing phases. To quote one of the highest-quality books in the field of machine and statistical learning, *Elements of Statistical Learning*, "It is difficult to give a general rule on how to choose the number of observations in each of the three parts." Fortunately, Hastie and friends immediately take pity on us poor practitioners and give a generic recommendation of 50%–25%–25% for training, validation testing, and held-out testing. That's about as good of a baseline split as we can get. With cross-validation, we could possibly consider a 75–25 split with 75% being thrown into the basket for cross-validation—which will be repeatedly split into training and validation-testing sets—and 25% saved away in a lockbox for final assessment. More on that shortly.

If we go back to the 50–25–25 split, let's drill into that 50%. We'll soon see evaluation tools called *learning curves*. These give us an indication of what happens to our validation-testing performance as we train on more and more examples. Often, at some high enough number of training examples, we will see a plateau in the performance. If that plateau happens within the 50% split size, things are looking pretty good for us. However, imagine a scenario where we need 90% of our available data to get a decent performance. Then, our 50–25–25 split is *simply not going to give a sufficiently good classifier* because we need more training data. We need a learner that is more efficient in its use of data.

5.2.2.4 Parameters and Hyperparameters

Now is the perfect time—I might be exaggerating—to deal with two other terms: parameters and hyperparameters. The knobs on a factory machine represent model parameters set by a learning method during the training phase. Choosing between different machines (3-NN or 10-NN) in the same overall class of machine (k-NN) is selecting a hyperparameter. Selecting hyperparameters, like selecting models, is done in the selection phase. Keep this distinction clear: *parameters* are set as *part of the learning method* in the training phase while *hyperparameters* are beyond the control of the learning method.

For a given run of a learning method, the available parameters (knobs) and the way they are used (internals of the factory machine) are fixed. We can only adjust the values those parameters take. Conceptually, this limitation can be a bit hard to describe. If the phases described above are talked about from outer to inner—in analogy with outer and inner

loops in a computer program—the order is *Assessment*, *Selection*, *Training*. Then, adjusting hyperparameters means stepping out one level from adjusting the parameters—stepping out from Training to Selection. We are thinking outside the box, if you will. At the same time—from a different perspective—we are *diving into* the inner workings of the machine like a mechanic. As is the case with rebuilding car engines, the training phase just doesn't go there.

With that perfect moment passed, we're going to minimize the discussion of hyperparameters for several chapters. If you want to know more about hyperparameters *right now*, go to Section 11.1. Table 5.1 summarizes the pieces we've discussed.

Table 5.1 Phases and datasets for learning.

Phase	Name	Dataset Used	Machine	Purpose
inner	training	training set	set knobs	optimize parameters
middle	selection	validation test set	choose machines	select model, hyperparameters
outer	assessment	hold-out test set	evaluate performance	assess future performance

For the middle phase, selection, let me emphasize just how easily we can mislead ourselves. We've only considered two kinds of classifiers so far: NB and k-NN. While k could grow arbitrarily big, we commonly limit it to relatively small values below 20 or so. So, maybe we are considering 21 total possible models (20 k-NN variants and 1 Naive Bayes model). Still, there are *many* other methods. In this book, we'll discuss about a half dozen. Several of these have almost *infinite* tunability. Instead of choosing between a k of 3, 10, or 20, some models have a C with any value from zero to infinity. Over many models and many tuning options, it is conceivable that we might hit the jackpot and find one combination that is *perfect* for our inner and middle phases. However, we've been indirectly peeking—homing in on the target by systematic guessing. Hopefully, it is now clear why the outer phase, assessment, is necessary to prevent ourselves from teaching to the test.

5.3 Major Tom, There's Something Wrong: Overfitting and Underfitting

Now that we've laid out some terminology for the learning phases—training, selection, and assessment—I want to dive into things that can go wrong with learning. Let's turn back to the exam scenario. *Mea culpa*. Suppose we take an exam and we don't do as well as we'd like. It would be nice if we could attribute our failure to something more specific than "bad, don't do that again." Two distinct failures are (1) not bringing enough raw

horsepower—capacity—to the exam and (2) focusing too much on irrelevant details. To align this story with our earlier discussion, number two is really just a case of being distracted by noise—but it makes us feel better about ourselves than binging on Netflix. These two sources of error have technical names: *underfitting* and *overfitting*. To investigate them, we're going to cook up a simple practice dataset.

5.3.1 Synthetic Data and Linear Regression

Often, I prefer to use real-world datasets—even if they are small—for examples. But in this case, we're going to use a bit of synthetic, genetically modified data. Creating synthetic data is a good tool to have in your toolbox. When we develop a learning system, we might need some data that we completely control. Creating our own data allows us to control the *true* underlying relationship between the inputs and the outputs and to manipulate how noise affects that relationship. We can specify both the type and amount of noise.

Here, we'll make a trivial dataset with one feature and a target, and make a train-test split on it. Our noise is chosen uniformly (you might want to revisit our discussion of distributions in Section 2.4.4) from values between −2 and 2.

In [2]:

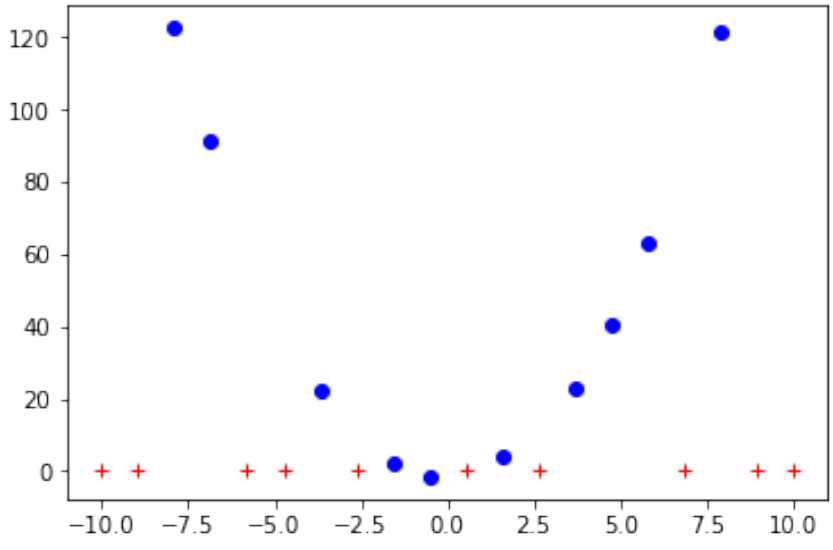
```
N = 20
ftr = np.linspace(-10, 10, num=N)                # ftr values
tgt = 2*ftr**2 - 3 + np.random.uniform(-2, 2, N) # tgt = func(ftr)

(train_ftr, test_ftr,
 train_tgt, test_tgt) = skms.train_test_split(ftr, tgt, test_size=N//2)

display(pd.DataFrame({"ftr":train_ftr,
                      "tgt":train_tgt}).T)
```

	0	1	2	3	4	5	6	7	8	9
ftr	-1.58	-6.84	-3.68	1.58	-7.90	3.68	7.89	4.74	5.79	-0.53
tgt	2.39	91.02	22.38	3.87	122.58	23.00	121.75	40.60	62.77	-1.61

Now we can take a look at that data visually. We have our known data points—the training set—in blue dots. The red pluses show the input feature values for the test set. We need to figure out how high up we should take each of those values.

In [3]:

```
plt.plot(train_ftr, train_tgt, 'bo')
plt.plot(test_ftr,  np.zeros_like(test_ftr), 'r+');
```



The numbers are a fairly straightforward example of a regression task. We have a numerical target value that we want to predict from an input. Now, we only have a few regression tools in our toolbox at this point. So, let's pull out linear regression (LR) and see what happens:

In [4]:

```
# note: sklearn *really* wants 2D inputs (a table)
# so we use rehape here.
sk_model = linear_model.LinearRegression()
sk_model.fit(train_ftr.reshape(-1, 1), train_tgt)
sk_preds = sk_model.predict(test_ftr.reshape(-1, 1))
sk_preds[:3]
```

Out[4]:

```
array([53.218 , 41.4552, 56.8374])
```

We're not evaluating these predictions in any way. But at least—like our training targets—they are positive values.

5.3.2 Manually Manipulating Model Complexity

Up until now, we've relied entirely on `sklearn` to do all the heavy lifting for us. Basically, `sklearn`'s methods have been responsible for setting the knob values on all the machines we've been using for our demonstrations. But there are *many* other packages for finding

those ideal knob values. Some of those packages are specifically geared towards machine learning. Others are geared towards specialized areas of mathematics and engineering.

One of those alternatives is the `polyfit` routine in NumPy. It takes input and output values, our features and a target, and a *degree of polynomial* to align with the data. It figures out the right knob values—actually, the coefficients of polynomials we discussed in Section 2.8—and then `np.poly1d` turns those coefficients into a function that can take inputs and produce outputs. Let's explore how it works:

In [5]:

```
# fit-predict-evaluate a 1D polynomial (a line)
model_one = np.poly1d(np.polyfit(train_ftr, train_tgt, 1))
preds_one = model_one(test_ftr)
print(preds_one[:3])
```

```
[53.218  41.4552 56.8374]
```

Interesting. The first three predictions are the same as our LR model. Are all of the predictions from these inputs the same? Yes. Let's demonstrate that and calculate the RMSE of the model:

In [6]:

```
# the predictions come back the same
print("all close?", np.allclose(sk_preds, preds_one))

# and we can still use sklearn to evaluate it
mse = metrics.mean_squared_error
print("RMSE:", np.sqrt(mse(test_tgt, preds_one)))
```

```
all close? True
RMSE: 86.69151817350722
```

Great. So, two take-home messages here. Message one: we can use alternative systems, not just `sklearn`, to learn models. We can even use those alternative systems *with* `sklearn` to do the evaluation. Message two: `np.polyfit`, as its name implies, can easily be manipulated to produce any degree of polynomial we are interested in. We have just fit a relatively simple line, but we can move beyond that to more complicated patterns. Let's explore that now.

One way to manipulate the complexity of linear regression is to ask, "What happens if we break out of our straight jacket and allow bends?" We can start answering that by looking at what happens when we add a single bend. For the non-mathphobic, a curve with one bend in it—called a parabola—is described by a degree-two polynomial. Instead of fitting a straight line to the points and picking the line with the lowest squared error, we're going to hold up parabolas—curves with a single bend—to the training data and find the one that fits best. The mathematics are surprisingly, or at least comfortingly, similar. As a result, our code only requires a minor tweak.

In [7]:

```
# fit-predict-evaluate a 2D polynomial (a parabola)
model_two = np.poly1d(np.polyfit(train_ftr, train_tgt, 2))
preds_two = model_two(test_ftr)
print("RMSE:", np.sqrt(mse(test_tgt, preds_two)))
```

```
RMSE: 1.2765992188881117
```

Hey, our test error improved quite a bit. Remember, error is like heat going out of your windows in the winter: we want very little of it! If one bend helped so well, maybe we just need a little more wiggle in our lives? Let's allow up to eight bends. If one was good, eight must be great! We can get eight bends from a degree-9 polynomial. You will really impress your dinner party guests if you tell them that a degree-9 polynomial is sometimes referred to as a *nonic*. Here's our degree-9 model's MSE:

In [8]:

```
model_three = np.poly1d(np.polyfit(train_ftr, train_tgt, 9))
preds_three = model_three(test_ftr)
print("RMSE:", np.sqrt(mse(test_tgt, preds_three)))
```

```
RMSE: 317.3634424235501
```

The error is significantly higher—worse—than we saw with the parabola. That might be unexpected. Let's investigate.

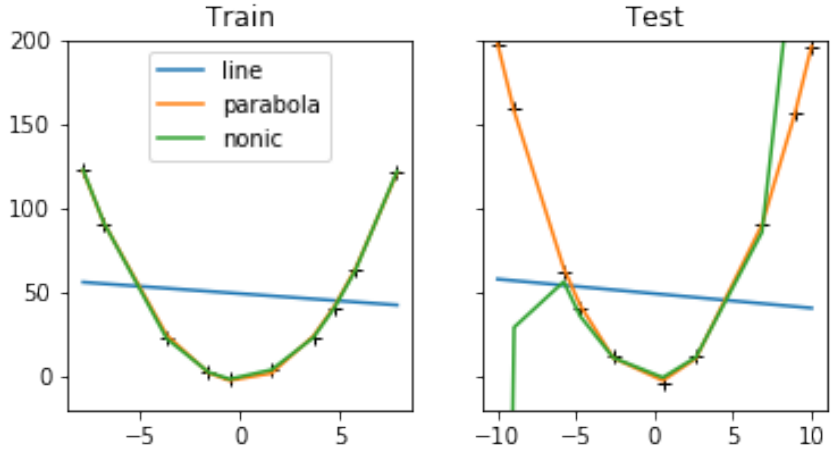
5.3.3 Goldilocks: Visualizing Overfitting, Underfitting, and "Just Right"

That didn't exactly go as planned. We didn't just get worse. We got utterly, terribly, horribly worse. What went wrong? We can break down what happened in the training and testing data visually:

In [9]:

```
fig, axes = plt.subplots(1, 2, figsize=(6, 3), sharey=True)

labels = ['line', 'parabola', 'nonic']
models = [model_one, model_two, model_three]
train = (train_ftr, train_tgt)
test  = (test_ftr, test_tgt)

for ax, (ftr, tgt) in zip(axes, [train, test]):
    ax.plot(ftr, tgt, 'k+')
```

```
    for m, lbl in zip(models, labels):
        ftr = sorted(ftr)
        ax.plot(ftr, m(ftr), '-', label=lbl)

axes[1].set_ylim(-20, 200)
axes[0].set_title("Train")
axes[1].set_title("Test");
axes[0].legend(loc='upper center');
```



`model_one`, the straight line, has great difficulty because our real model follows a curved trajectory. `model_two` eats that up: it follows the curve just about perfectly. `model_three` *seems* to do wonderfully when we train. It basically overlaps with both `model_two` and the real outputs. However, it has problems when we go to testing. It starts exploding out of control near `ftr=-7`. For ease of comparison, we can rerun the models and gather up the results in one table. Since it is easy to add another midway model, I'll also throw in a degree-6 model.

In [10]:

```
results = []
for complexity in [1, 2, 6, 9]:
    model = np.poly1d(np.polyfit(train_ftr, train_tgt, complexity))
    train_error = np.sqrt(mse(train_tgt, model(train_ftr)))
    test_error = np.sqrt(mse(test_tgt, model(test_ftr)))
    results.append((complexity, train_error, test_error))
```

```
columns = ["Complexity", "Train Error", "Test Error"]
results_df = pd.DataFrame.from_records(results,
                                       columns=columns,
                                       index="Complexity")

results_df
```

Out[10]:

Complexity	Train Error	Test Error
1	45.4951	86.6915
2	1.0828	1.2766
6	0.2819	6.1417
9	0.0000	317.3634

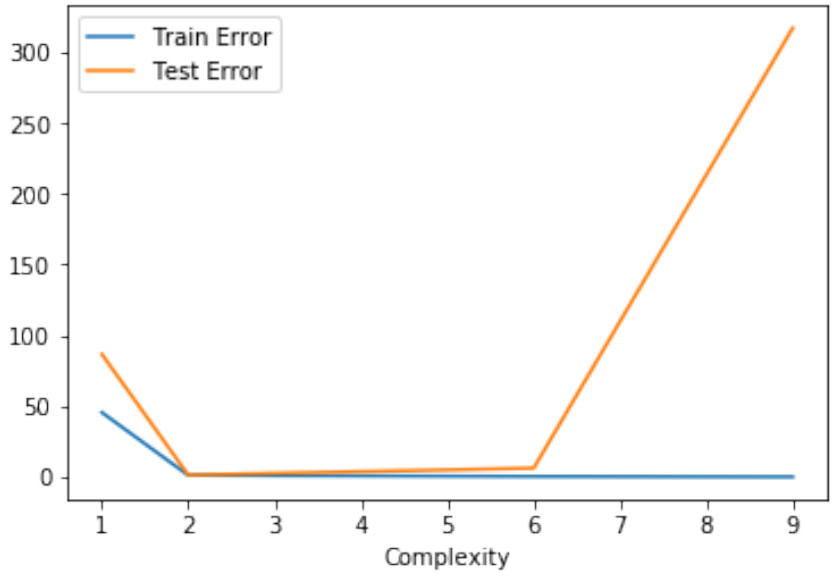
Let's review what happened with each of the three models with complexity 1, 2, and 9.

- Model one (Complexity 1—a straight line). Model one was *completely* outclassed. It brought a tricycle to a Formula One race. It was doomed from the beginning. The model doesn't have enough raw horsepower, or capacity, to capture the complexity of a target. It is too biased towards flatness. The model is *underfitting*.
- Model three (Complexity 9—a wiggly 9-degree polynomial). Model three certainly had enough horsepower. We see that it does very well on the training data. In fact, it gets to the point where it is *perfect* on the training data. But it completely falls apart when it comes to testing. Why? Because it memorized the *noise*—the randomness in the data. It varies too much with the data. We call this *overfitting*.
- Model two (Complexity 2—a parabola). Here we have the Goldilocks solution: it's not too hot, it's not too cold, it's just right. We have enough horsepower, but not so much that we can't control it. We do well enough on the training data and we see that we are at the lowest *testing* error. If we had set up a full validation step to select between the three machines with different complexity, we would be quite happy with model two. Model two doesn't *exactly* capture the training patterns because the training patterns include *noise*.

Let's graph out the results on the train and test sets:

In [11]:

```
results_df.plot();
```



The key take-away from the graph is that as we ratchet up the complexity of our model, we get to a point where we can make the training error very, very small—perhaps even zero. It is a Pyrrhic victory. Where it really counts—on the test set—we get worse. Then, we get terrible. We get to give up and go home bad. To highlight the important pieces of that graph, a version with helpful labels is shown in Figure 5.4.

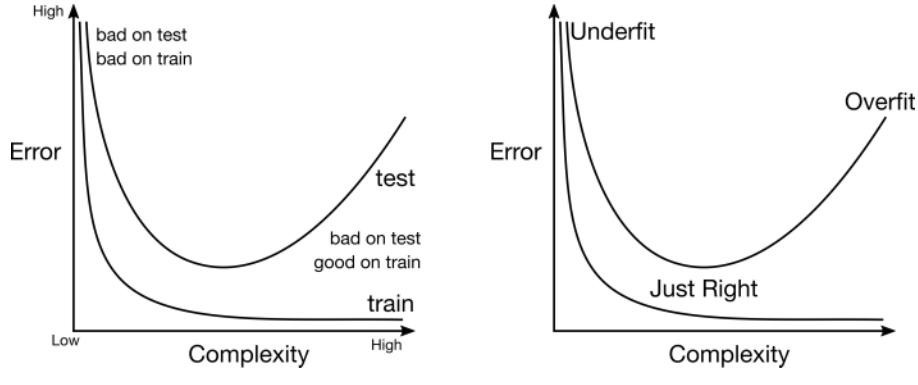


Figure 5.4 As complexity increases, we generally move from underfitting to just right to overfitting.

5.3.4 Simplicity

Let's spend one more minute talking about complexity and its good friend, simplicity. We just saw a concrete example of added complexity making our performance worse. That's because the added complexity wasn't used for the right reasons. It was spent following the noise instead of the true pattern. We don't really get to choose how complexity is used by our learners. They *have* a complexity—which can be used for good or bad. So, if we had several learners that all performed the same but had differing complexities, the potential abuse of power—by which I mean complexity—might lead us to prefer the simplest of these equal-performing models.

The underlying idea—simplicity is an important rule-of-thumb—is known throughout science and philosophy as Occam's razor (Ockham's razor for you historians) from a quote by William of Ockham, "Entities are not to be multiplied without necessity." (That's translated from Latin.) For us, the message is that we don't want more complexity in our model unless there is a reason. Put concretely, we don't want a higher-degree polynomial unless it pays us back in lower *test* error. Paraphrasing a much longer actual quote from Einstein, "Make things as simple as possible, but no simpler." (Hey, if I'm going to name-drop, I'm not going to stop at a philosopher who lived around 1300!)

Here's a thought that might keep you awake tonight. There is a learning method, which we will discuss later in Section 12.4, that can continue improving its test set performance even after it has *apparently* mastered the training set. Students of machine learning say it has driven the training error to zero but it is still improving. What's a real-life equivalent? You might imagine smoothing out your own rough edges in a public performance. Even after you've completed rehearsing a scene or preparing a dish well enough for friends and family, there is more you can do before you are ready for the public. Before your opening night on stage, you want to be *on point* for the harshest critic. Amazingly, there's a learning system that can transition from a friends-and-family rehearsal to a Broadway show.

5.3.5 Take-Home Notes on Overfitting

This section has a lot to consider. Here are the key points:

- Underfitting: A very simple model may not be able to learn the pattern in the training data. It also does poorly on the testing data.
- Overfitting: A very complex model may learn the training data perfectly. However, it does poorly on the testing data because it also learned irrelevant relationships in the training data.
- Just-right: A medium-complexity model performs well on the training and testing data.

We need the right tradeoff between simplicity and complexity to find a just-right model.

5.4 From Errors to Costs

In our discussion of overfitting and underfitting, we compared model complexity and error rates. We saw that as we vary the complexity of a class of models—the degree of our polynomials—we have different training and test performance. These two aspects, error and complexity, are intimately tied together. As we wander deeper into the zoo of learning methods, we'll see that some methods can explicitly trade off training error for complexity. In terms of our factory machine, we can consider *both* our success in copying the input-output relationship *and* the values we set on the knobs. Separating out these two aspects of "model goodness" lets us speak pretty generally about both regression and classification problems. When we progress a bit more, we'll also be able to *describe* many different algorithms in terms of a just a few choices. How a method treats errors and complexity are two of those choices.

5.4.1 Loss

So, what is our breakdown? First, we'll construct a *loss function* that quantifies what happens when our model is wrong on a single example. We'll use that to build a *training loss* function that measures how well our model does on the entire training set. More technical write-ups call this the *empirical loss*. Empirical simply means "by observation" or "as seen," so it's a loss based on the data we've seen. The training loss is the sum of the losses on each example. We can write that in code as:

In [12]:

```
def training_loss(loss, model, training_data):
    ' total training_loss on train_data with model under loss'
    return sum(loss(model.predict(x.reshape(1, -1)), y)
                                for x, y in training_data)
def squared_error(prediction, actual):
    ' squared error on a single example '
    return (prediction - actual)**2

# could be used like:
# my_training_loss = training_loss(squared_error, model, training_data)
```

A generic mathematical way to write this is:

$$\text{TrainingLoss}_{\text{Loss}}(m, D_{\text{train}}) = \sum_{x,y \in D_{\text{train}}} \text{Loss}\big(m(x), y\big)$$

and for the specific case of squared-error (SE) on 3-NN, where 3-NN(x) represents the prediction of 3-NN on an example x:

$$\text{TrainingLoss}_{\text{SE}}(\text{3-NN}, D_{\text{train}}) = \sum_{x,y \in D_{\text{train}}} \text{SE}\big(\text{3-NN}(x), y\big) = \sum_{x,y \in D_{\text{train}}} \big(\text{3-NN}(x) - y\big)^2$$

We can put that to use with:

In [13]:

```
knn = neighbors.KNeighborsRegressor(n_neighbors=3)
fit = knn.fit(diabetes.data, diabetes.target)

training_data = zip(diabetes.data, diabetes.target)

my_training_loss = training_loss(squared_error,
                                 knn,
                                 training_data)
print(my_training_loss)
```

```
[863792.3333]
```

If we use `sklearn`'s `mean_squared_error` and multiply it by the number of training examples—to undo the `mean` part—we get the same answer.

In [14]:

```
mse = metrics.mean_squared_error(diabetes.target,
                                 knn.predict(diabetes.data))
print(mse*len(diabetes.data))
```

```
863792.3333333333
```

The somewhat scary equation for TraingingLoss is a fundamental principle that underlies the evaluation calculations we use. We will also add on to that equation, literally, to deal with the problem of determining a good model complexity.

5.4.2 Cost

As we saw with overfitting, if we make our model more and more complex, we can capture *any* pattern—even pattern that is really noise. So, we need something that works against complexity and rewards simplicity. We do that by adding a value to the training loss to create a total notion of *cost*. Conceptually, cost = loss + complexity, but we have to fill in some details. The term we add to deal with complexity has several technical names: regularization, smoothing, penalization, or shrinkage. We'll just call it *complexity*. In short, the total cost we pay to use a model on some data depends on (1) how well it does and (2) how complicated it is. You can think of the complexity part as a baseline investment. If we have a very high initial investment, we better not have many errors. Conversely, if we have a low initial investment, we might have some room to allow for error. All of this is because we want good performance on unseen data. The term for performance on novel, unseen data is *generalization*.

One last comment. We don't have to have a fixed idea of how to trade off error and complexity. We can leave it as an open question and it will become part of the way our

machine is built. In technical terms, it's just another hyperparameter. To use a traditional naming scheme—and to help break mathphobias—I'm going to use a lower-case Greek letter, λ, pronounced "lamb-da" as in "it's a lamb, duh." Lambda represents that tradeoff. While it can be unnatural to phrase some learners strictly in terms of loss and complexity, it is very broadly possible. We'll discuss that idea more in Chapter 15. We can choose a good value of λ by performing several rounds of validation testing and taking the λ that leads to the lowest cost.

In [15]:

```
def complexity(model):
    return model.complexity

def cost(model, training_data, loss, _lambda):
    return training_loss(m,D) + _lambda * complexity(m)
```

Mathematically, that looks like

$$\text{Cost}(m, D_{\text{train}}, \text{Loss}, \lambda) = \text{TrainingLoss}_{\text{Loss}}(m, D_{\text{train}}) + \lambda \text{Complexity}(m)$$

That is, our cost goes up (1) if we make more mistakes and (2) if we invest resources in more expensive, but also more flexible, models. If we take $\lambda = 2$, one unit of complexity is comparable to two units of loss. If we take $\lambda = .5$, two units of complexity are comparable to one unit of loss. Shifting λ adjusts how much we care about errors and complexity.

5.4.3 Score

You will also see the term *score* or *scoring function*. Scoring functions—at least in `sklearn`'s lexicon—are a variation on quantifying loss where bigger values are better. For our purpose, we can consider losses and scores to be inverses: as one goes up, the other goes down. So, we generally want a *high score* or a *low loss*. It simply depends on which sort of measurement we are using; they are two different ways of saying the same thing. Another set of opposites is that we will want to *minimize* a loss or loss function but we will *maximize* a score or scoring function. To summarize:

- Score: higher is better, try to maximize.
- Loss, error, and cost: lower is better, try to minimize.

Once again, if we have two models, we can compare their costs. If we have many different models, we can use some combination of brute force, blind or clever search, and mathematical trickery to pick the lowest-cost models among those—we discussed these alternatives in Section 4.4. Of course, we might be wrong. There might be models we didn't consider that have even lower cost. Our cost might not be the ideal way of evaluating the models' performance in the real world. Our complexity measure, or our tradeoff for complexity, might be too high or too low. All of these factors are working behind the scenes when we haltingly say, "We picked the *best* model and hyperparameters." Well, we did, at least up to the guesses, assumptions, and constraints we used.

We'll discuss the practical side of picking good, or at least better, model hyperparameters in Section 11.2. As we will see, modern machine learning software such as `sklearn` makes it *very easy* to try many models and combinations of hyperparameters.

5.5 (Re)Sampling: Making More from Less

If we content ourselves with a single train-test split, that single step provides and determines both the data we can train from and our testing environment. It is a simple method, for sure. However, we might get (un)lucky and get a very *good* train-test split. Imagine we get very *hard* training data and very *easy* testing data. Boom—all of a sudden, we are *overestimating* how well we will do in the big, bad real world. If overconfidence is our real concern, we'd actually like a worst-case scenario: easy training and hard testing that would lead us to underestimate the real-world performance. Enough with single-evaluation scenarios, however. Is there a way we could do better? If we ask people to estimate the number of beans in a jar, they will individually be wrong. But if we get many, many estimates, we can get a better overall answer. Hurray for the wisdom of crowds. So, how do we generate multiple estimates for evaluation? We need multiple datasets. But we only have one dataset available. How can we turn one dataset into many?

5.5.1 Cross-Validation

The machine learning community's basic answer to generating multiple datasets is called *cross-validation*. Cross-validation is like a card game where we deal out all the cards to three players, play a round of the game, and then shift our cards to the player on the right—and repeat that process until we've played the game with each of the three different sets of cards. To figure out our overall score, we take the individual scores from each different hand we played with and combine them, often with an average.

Let's go directly to an example. Cross-validation takes a number of *folds* which is like the number of players we had above. With three players, or three folds, we get three different attempts to play the game. For 3-fold cross-validation, we'll take an entire set of labeled data and shake it up. We'll let the data fall randomly—as evenly as possible—into the three buckets in Figure 5.5 labeled with Roman numerals: B_I, B_{II}, and B_{III}.

Now, we perform the following steps shown in Figure 5.6:

1. Take bucket B_I and put it to the side. Put B_{II} and B_{III} together and use them as our training set. Train a model—we'll call it *ModelOne*—from that combined training set. Now, evaluate *ModelOne* on bucket B_I and record the performance as *EvalOne*.
2. Take B_{II} and put it to the side. Put buckets B_I and B_{III} together and use them as our training set. Train *ModelTwo* from that combined training set. Now, evaluate *ModelTwo* on bucket B_{II} and record the performance as *EvalTwo*.

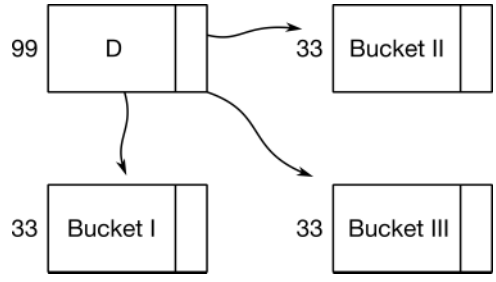


Figure 5.5 Our setup for cross-validation splits the data into approximately equal buckets.

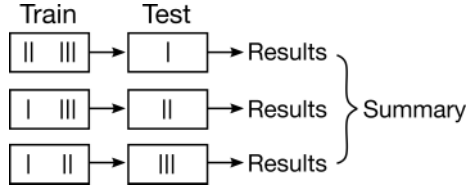


Figure 5.6 We use each cross-validation bucket, in turn, as our test data. We train on the remainder.

3. Take bucket B_{III} and put it to the side. Put B_I and B_{II} together as our training set. Train *ModelThree* from that combined training set. Now, evaluate *ModelThree* on B_{III} and record the performance as *EvalThree*.

Now, we've recorded three performance values. We can do several things with the values, including graphing them and summarizing them statistically. Graphing them can help us understand how variable our performance is with respect to different training and testing datasets. It tells us something about how our model, our training sets, and our testing sets interact. If we see a very wide spread in our performance measures, we would be justified in being skeptical of any single performance score for our system. On the other hand, if the scores are all similar, we have some certainty that, regardless of the specific train-test split, our system's performance will be similar. One caveat: our random sampling is done without replacement and the train-test splits are all dependent on each other. This breaks some of the usual assumptions we make in statistics land. If you are concerned about it, you might want to see Section 5.5.3.

What we've described is 3-fold cross-validation. The general name for CV techniques is *k*-fold cross-validation—I'll usually abbreviate it as *k*-fold CV or just *k*-CV. The amount of cross-validation we do depends on a few factors including the amount of data we have. 3-, 5-, and 10-fold CV are commonly used and recommended. But we're getting ahead of ourselves. Here's a simple example of *5*-fold CV with `sklearn`:

In [16]:

```
# data, model, fit & cv-score
model = neighbors.KNeighborsRegressor(10)
skms.cross_val_score(model,
                     diabetes.data,
                     diabetes.target,
                     cv=5,
                     scoring='neg_mean_squared_error')
# notes:
# defaults for cross_val_score are
# cv=3 fold, no shuffle, stratified if classifier
# model.score by default (regressors: r2, classifiers: accuracy)
```

Out[16]:

```
array([-3206.7542, -3426.4313, -3587.9422, -3039.4944, -3282.6016])
```

The default value for the `cv` argument to `cross_val_score` is `None`. To understand what that means, we have to look into the documentation for `cross_val_score`. Here is the relevant part, cleaned up and simplified a bit:

> `cv`: int or None or others. Determines the cross-validation splitting strategy. Possible inputs for cv are:
>
> - None, to use the default 3-fold cross validation,
> - Integer, to specify the number of folds in a `(Stratified)KFold`,
> - Others.
>
> For integer/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, `StratifiedKFold` is used. In all other cases, `KFold` is used.

We haven't discussed stratification yet—we'll get to it after the next commercial break—but there are two take-home lessons: (1) by default we're doing 3-fold CV and (2) for classification problems, `sklearn` uses stratification.

So, what's up with the `scoring='neg_mean_squared_error'` argument? You might recall mean squared error (MSE) being a thing. You are in the right ballpark. However, we have to reconcile "error up, bad" with "score up, good." To do that, `sklearn` negates the MSE to go from an error measure to a score. The scores are all *negative*, but a bigger score is better. Think of it as losing less money: instead of being down \$100, you are only down \$7.50.

One last bit on regression and scoring: the default scoring for regressors is `r2` (R^2 for the math folks). It is well known in statistics under the name *coefficient of determination*. We'll discuss it in Section 7.2.3 but for now, let me simply say that *it is very, very easy to misuse and abuse* R^2. You may be carrying R^2 baggage with you—please leave it at the door. This is not your classic R^2.

We haven't talked about what value of k to use. The dirtiest secret of k-CV is that we need to balance three things. The first issue: how long does it take to train and test our

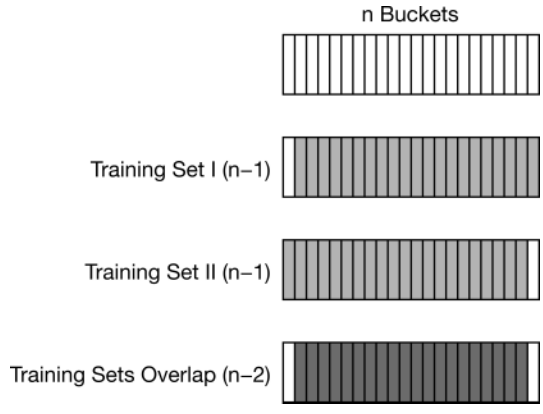


Figure 5.7 Overlap in n-fold cross-validation.

models? A bigger k means more buckets which in turn means more training and testing phases. There are some *incremental/decremental* learning methods that can be implemented to minimize the amount of work necessary to train and test different models. Unfortunately, most common models are *not* implemented or interfaced with cross-validation routines in a way that allows this efficiency. To make matters even more interesting, training on smaller datasets is *faster* than training on larger datasets. So, we are comparing many small trainings (and large testings) with fewer larger trainings (and smaller testings). The run times will depend on the specific learning methods you use.

The second issue to balance is that the value of k slides up and down between two extremes. The smallest useful value is $k = 2$, which makes two buckets of data and two estimates of our test error. The largest value k can take is the number of data points we have, $k = n$. This results in each example being in its own bucket and n total models and estimates. In addition, when we have two buckets, we never train on the same data. With three buckets, we have some overlap—half the training set used to create the model tested on B_I is in common with the training sets used to create the model tested on B_{II}. Specifically, the common elements are those in B_{III}. With n buckets, between any two of our models, they are trained on the *same* $n - 2$ examples (see Figure 5.7). To get to the full n examples, there's one more example for training that is different in the two CV folds and another example that is reserved for testing.

The net effect is that the data in the training folds for $k = 2$ is *very different* and the data in the training folds for $k = n$ is *almost the same*. This means the estimates we get out of $k = 2$ will be quite different—if there is a difference to be found. The estimates out of $k = n$ will be very similar, because they are doing *almost* the same thing!

The third issue is that small values of k—relatively few folds—will result in training set sizes ranging from 50% ($k = 2$) to 90% ($k = 10$) of the data. Whether this is acceptable—whether learning on that much data will be sufficient—depends on the problem at hand. We can evaluate that graphically using learning curves, as in Section 5.7.1. If the learning curve flattens out at the percent of data we are trying to learn from—if that much data is enough to get us to a sufficient performance threshold—we are probably OK using the related number of folds.

5.5.2 Stratification

Let's turn to a quick example of cross-validation in a classification context. Here, we tell `cross_val_score` to use 5-fold CV.

In [17]:

```
iris = datasets.load_iris()
model = neighbors.KNeighborsClassifier(10)
skms.cross_val_score(model, iris.data, iris.target, cv=5)
```

Out[17]:

```
array([0.9667, 1.    , 1.    , 0.9333, 1.    ])
```

As mentioned above, the cross-validation was done in a *stratified* manner because it is the default for classifiers in `sklearn`. What does that mean? Basically, stratification means that when we make our training-testing splits for cross-validation, we want to respect the proportions of the targets that are present in our data. Huh? Let's do an example. Here's a tiny dataset targeting cats and dogs that we'll take two-fold training samples from:

In [18]:

```
# not stratified
pet = np.array(['cat', 'dog', 'cat',
                'dog', 'dog', 'dog'])
list_folds = list(skms.KFold(2).split(pet))
training_idxs = np.array(list_folds)[:, 0, :]

print(pet[training_idxs])
```

```
[['dog' 'dog' 'dog']
 ['cat' 'dog' 'cat']]
```

Out cat-loving readers will notice that there were no `cat`s in the first fold. That's not great. If that were our target, we would have no examples to learn about cats. Simply put, that can't be good. Stratified sampling enforces fair play among the cats and dogs:

In [19]:

```
# stratified
# note: typically this is behind the scenes
# making StratifiedKFold produce readable output
# requires some trickery.  feel free to ignore.
pet = np.array(['cat', 'dog', 'cat', 'dog', 'dog', 'dog'])
```

```
idxs = np.array(list(skms.StratifiedKFold(2)
                         .split(np.ones_like(pet), pet)))
training_idxs = idxs[:, 0, :]
print(pet[training_idxs])
```

```
[['cat' 'dog' 'dog']
 ['cat' 'dog' 'dog']]
```

Now, both folds have a balanced number of cats and dogs, equal to their proportion in the overall dataset. Stratification ensures that we have the same (or nearly the same, once we round off uneven splits) *percent* of dogs and cats in each of our training sets as we do in our entire, available population. Without stratification, we could end up having too few (or even none) of a target class—in our nonstratified example, the first training set had no cats. We don't expect that training data to lead to a good model.

Stratification is particularly useful when (1) we have limited data overall or (2) we have classes that are poorly represented in our dataset. Poor representation might be due to rareness—if we're talking about an uncommon disease or winning lottery tickets—or it might be due to our data collection processes. Having a limited total amount of data makes *everything* rare, in a sense. We will discuss more issues around rare classes in Section 6.2.

How does the default stratification apply to the *iris* dataset in Chapter 3? It means that when we perform the cross-validation splits, we can be sure that each of the training sets has a balanced representation from each of the three possible target flowers. What if we don't want stratification? It's slightly more tricky, but we can do it:

In [20]:

```
# running nonstratified CV
iris = datasets.load_iris()
model = neighbors.KNeighborsClassifier(10)
non_strat_kf = skms.KFold(5)
skms.cross_val_score(model,
                     iris.data,
                     iris.target,
                     cv=non_strat_kf)
```

Out[20]:

```
array([1.    , 1.    , 0.8667, 0.9667, 0.7667])
```

We can make an educated guess that the last fold probably had a bad distribution of flowers. We probably didn't see enough of one of the species to learn patterns to identify it.

5.5.3 Repeated Train-Test Splits

Here's another example of the train-test split with an added twist. (That's a train-test twist, if you're keeping track.) Our twist is that we are going to do some repeated coin flipping

to generate several train-test splits. Why do we want to *repeat* the fundamental train-test split step? Any time we rely on randomness, we are subject to variation: several different train-test splits might give different results. Some of those might turn out wonderfully and some may turn out horribly. In some scenarios, like playing the lottery, the *vast majority* of outcomes are very similar—you don't win money. In others, we don't know ahead of time what the outcomes are. Fortunately, we have an extremely useful tool at our disposal that we can pull out when confronted with unknown randomness. Do the random thing many times and see what happens. Stand back, we're about to try science!

In the case of train-test splits, we generally don't know ahead of time how well we expect to perform. Maybe the problem is really easy and almost all of the train-test splits will give a good learner that performs well on the test set. Or maybe it is a very hard problem and we happen to select an easy subset of training data—we do great in training, but perform horribly in testing. We can investigate the variation due to the train-test split by making many train-test splits and looking at different results. We do that by randomly resplitting several times and evaluating the outcomes. We can even compute statistics—the mean, median, or variance—of the results if we really want to get technical. However, I am always a fan of *looking at the data* before we get into *summarizing the data* with statistics.

The multiple values—one per train-test split—get us a distribution of the results and how often they occur. Just like drawing a graph of the heights of students in a classroom gets us a distribution of those heights, repeated train-test splits get us a distribution of our evaluation measure—whether it is accuracy, root-mean-squared-error, or something else. The distribution is *not* over *every* possible source of variation. It is simply taking into account one difference due to randomness: how we picked the training and testing data. We can see how *variable* our result is *due to* the randomness of making a train-test split. Without further ado, let's look at some results.

In [21]:

```
# as a reminder, these are some of the imports
# that are hidden behind: from mlwpy import *
# from sklearn import (datasets, neighbors,
#                      model_selection as skms,
#                      linear_model, metrics)
# see Appendix A for details

linreg   = linear_model.LinearRegression()
diabetes = datasets.load_diabetes()

scores = []
for r in range(10):
    tts = skms.train_test_split(diabetes.data,
                                diabetes.target,
                                test_size=.25)
```

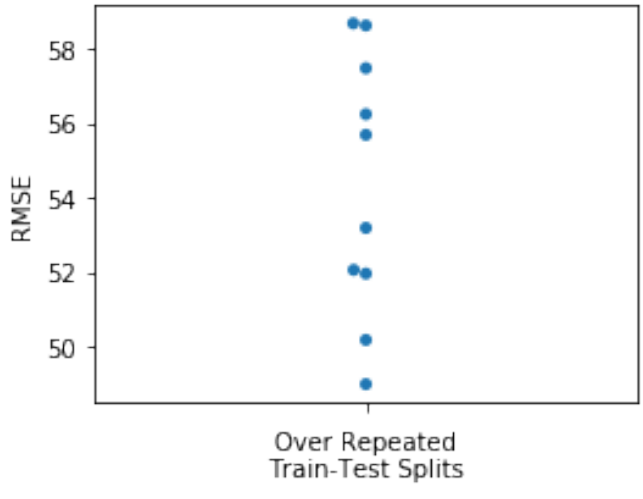
```
    (diabetes_train_ftrs, diabetes_test_ftrs,
     diabetes_train_tgt,  diabetes_test_tgt) = tts

    fit   = linreg.fit(diabetes_train_ftrs, diabetes_train_tgt)
    preds = fit.predict(diabetes_test_ftrs)

    score = metrics.mean_squared_error(diabetes_test_tgt, preds)
    scores.append(score)

scores = pd.Series(np.sqrt(sorted(scores)))
df = pd.DataFrame({'RMSE':scores})
df.index.name = 'Repeat'
display(df.T)
```

Repeat	0	1	2	3	4	5	6	7	8	9
RMSE	49.00	50.19	51.97	52.07	53.20	55.70	56.25	57.49	58.64	58.69

You can certainly take *looking at the data* to an extreme. A raw list is only useful for relatively few values—people don't scale well to reading too many numbers. Let's make a plot. `swarmplot`, from the Seaborn library, is very useful here. It makes a single value plot—also called a stripplot—and stacks repeated values horizontally so you get a feel for where there are clumps of values.

In [22]:

```
ax = plt.figure(figsize=(4, 3)).gca()
sns.swarmplot(y='RMSE', data=df, ax=ax)
ax.set_xlabel('Over Repeated\nTrain-Test Splits');
```



In [23]:

```
display(df.describe().T)
```

	count	mean	std	min	25%	50%	75%	max
RMSE	10.000	54.322	3.506	49.003	51.998	54.451	57.182	58.694

When evaluating plots like this, always orient yourself to the scale of the data. At first, we might think that this data is pretty spread out, but upon further review, we see that it is clustered in the mid-to-upper 50s. Whether that is "a lot" depends on size of the RMSE values—the mean is near 55, so we are in the ballpark of ±10%. That's large enough to warrant our attention.

As a quick Python lesson, here's a way we can rewrite the score-computing code above with a list comprehension instead of a loop. The basic strategy is to (1) take the contents of the loop and turn it into a function and (2) use that function repeatedly in a list comprehension. This rewrite gets us a bit of performance gain—but I'm not doing it for the resource optimization. The biggest win is that we've given a *name* to our process of making a train-test split, fitting, predicting, and evaluating. As in the book of Genesis, naming is one of the most powerful things we can do in a computer program. Defining a function also gives us a single entity that we can test for resource use and *reuse* in other code.

In [24]:

```
def tts_fit_score(model, data, msr, test_size=.25):
    ' apply a train-test split to fit model on data and eval with MSR '
    tts = skms.train_test_split(data.data,
                                data.target,
                                test_size=test_size)

    (train_ftrs, test_ftrs, train_tgt,  test_tgt) = tts

    fit   = linreg.fit(train_ftrs, train_tgt)
    preds = fit.predict(test_ftrs)

    score = msr(test_tgt, preds)
    return score

linreg   = linear_model.LinearRegression()
diabetes = datasets.load_diabetes()
scores = [tts_fit_score(linreg, diabetes,
                        metrics.mean_squared_error) for i in range(10)]
print(np.mean(scores))
```

```
3052.540273057884
```

I'll leave you with one final comment on repeated train-test splits and cross-validation. With *k*-CV, we will get one, and only one, prediction for each and every example. Each example is in precisely one test bucket. The predictions for the *whole* dataset will be aggregated from the *k* models that are developed on different sets of data. With repeated train-test splits, we may completely ignore training or predicting on some examples and make repeated predictions on other examples as we see in Figure 5.8. In repeated train-test splits, it is all subject to the randomness of our selection process.

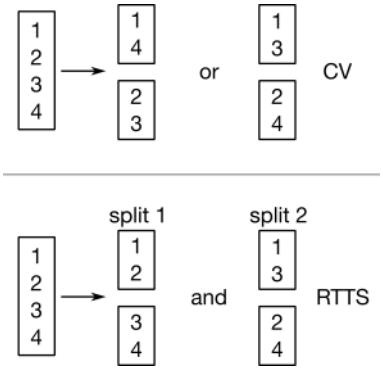


Figure 5.8 RTTS may have duplication between repeats.

5.5.4 A Better Way and Shuffling

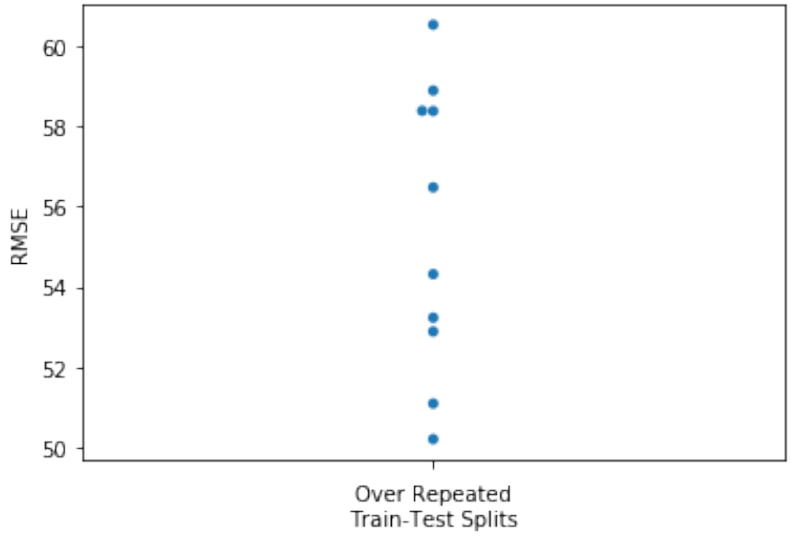
Managing the repeated looping to make multiple train-test splits was a bit annoying. It was not heart- or back-breaking, but there are many places we could make a mistake. It would be nice if someone wrapped the process up in a single stand-alone function. Fortunately, `sklearn` has done that. If we pass in a `ShuffleSplit` data-splitter to the `cv` argument of `cross_val_score`, we get precisely the algorithm we hand-coded above.

In [25]:

```
linreg   = linear_model.LinearRegression()
diabetes = datasets.load_diabetes()

# nondefault cv= argument
ss = skms.ShuffleSplit(test_size=.25) # default, 10 splits
scores = skms.cross_val_score(linreg,
                              diabetes.data, diabetes.target,
                              cv=ss,
                              scoring='neg_mean_squared_error')

scores = pd.Series(np.sqrt(-scores))
df = pd.DataFrame({'RMSE':scores})
df.index.name = 'Repeat'
```

```
display(df.describe().T)

ax = sns.swarmplot(y='RMSE', data=df)
ax.set_xlabel('Over Repeated\nTrain-Test Splits');
```

	count	mean	std	min	25%	50%	75%	max
RMSE	10.000	55.439	3.587	50.190	52.966	55.397	58.391	60.543



The slight differences with our manual version are due to randomly selecting the train-test splits.

Now, I want to talk about another way that randomness affects us as intrepid students of machine learning. It's the kind of randomness that the computer uses when we ask it to do random things. Here's what's going on behind the scenes with `ShuffleSplit`. Don't worry, I'll explain `random_state` in just a second.

In [26]:

```
ss = skms.ShuffleSplit(test_size=.25, random_state=42)

train, test = 0, 1
next(ss.split(diabetes.data))[train][:10]
```

Out[26]:

```
array([ 16, 408, 432, 316,   3,  18, 355,  60, 398, 124])
```

By the way, I use `next` because `ShuffleSplit` relies on a Python generator to produce one split after another. Saying `next` provides me with the next data split. After fetching the next data split, I pick out the training data `[train]` and then the first ten examples `[:10]`.

Good enough. Let's try again.

In [27]:

```
ss = skms.ShuffleSplit(test_size=.25, random_state=42)
next(ss.split(diabetes.data))[train][:10]
```

Out[27]:

```
array([ 16, 408, 432, 316,   3,  18, 355,  60, 398, 124])
```

That can't be good. Someone call—someone! We need help. The results are the same. Wasn't this supposed to be random? The answer is yes . . . and no. Randomness on a computer is often pseudo-random. It's a long list of numbers that, when put together, are random enough to fake their randomness. We start at some point in that list and start taking values. To an outside observer, they seem pretty random. But if you know the mechanism, you could actually know what values are coming ahead of time. Thus, (1) the values we generate will *look* mostly random, but (2) the process used to generate them is actually deterministic. This determinism has a nice side effect that we can take advantage of. If we specify a starting point for the sequence of pseudo-random numbers, we can get a reproducible list of the not-so-random values. When we use `random_state`, we are setting a starting point for `ShuffleSplit` to use when it asks for randomness. We'll end up getting the same outputs. Repeatable train-test splitting is very useful for creating reproducible test cases, sharing examples with students, and eliminating some degrees of freedom when chasing down bugs.

While we're at it, here's another place where a similar issue comes up. Let's do two separate runs of `KFold`ing.

In [28]:

```
train, test = 0, 1
kf = skms.KFold(5)
next(kf.split(diabetes.data))[train][:10]
```

Out[28]:

```
array([89, 90, 91, 92, 93, 94, 95, 96, 97, 98])
```

In [29]:

```
kf = skms.KFold(5)
next(kf.split(diabetes.data))[train][:10]
```

Out[29]:

```
array([89, 90, 91, 92, 93, 94, 95, 96, 97, 98])
```

The lack of randomness, in places we *want* randomness, is starting to get a little old. The issue here is the default parameters to `KFold`:

```
skms.KFold(n_splits=3, shuffle=False, random_state=None)
```

`shuffle=False`, the default, means that we *don't* shake up the examples before distributing them to different folds. If we *want* them shaken up, we need to say so. To keep the examples a bit more readable, we'll switch back to the simple `pet` targets.

In [30]:

```
pet = np.array(['cat', 'dog', 'cat',
                'dog', 'dog', 'dog'])

kf = skms.KFold(3, shuffle=True)

train, test = 0, 1
split_1_group_1 = next(kf.split(pet))[train]
split_2_group_1 = next(kf.split(pet))[train]

print(split_1_group_1,
      split_2_group_1)
```

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

If we set a `random_state`, it's shared by the splitters:

In [31]:

```
kf = skms.KFold(3, shuffle=True, random_state=42)

split_1_group_1 = next(kf.split(pet))[train]
split_2_group_1 = next(kf.split(pet))[train]
print(split_1_group_1,
      split_2_group_1)
```

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

5.5.5 Leave-One-Out Cross-Validation

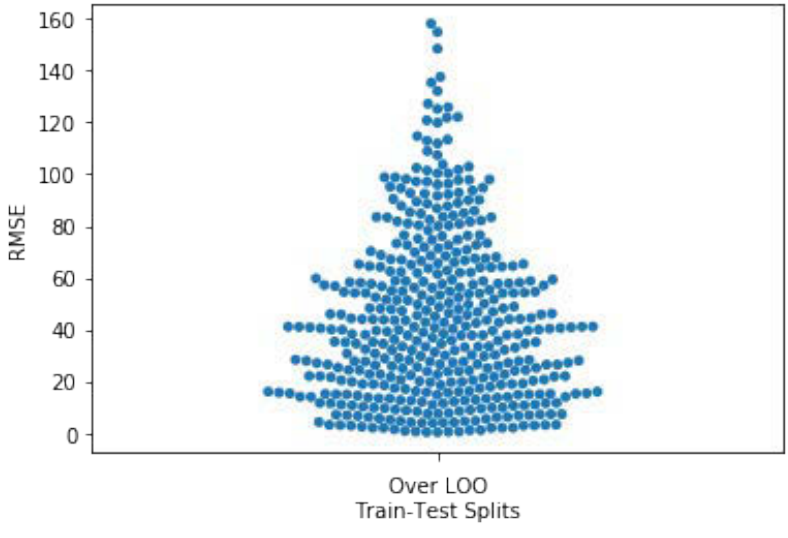
I mentioned above that we could take an extreme approach to cross-validation and use as many cross-validation buckets as we have examples. So, with 20 examples, we could potentially make 20 train-test splits, do 20 training fits, do 20 testing rounds, and get 20

resulting evaluations. This version of CV is called *leave-one-out cross-validation* (LOOCV) and it is interesting because *all* of the models we generate are going to have *almost all* of their training data in common. With 20 examples, 90% of the data is shared between any two training runs. You might refer back to Figure 5.7 to see it visually.

In [32]:

```
linreg   = linear_model.LinearRegression()
diabetes = datasets.load_diabetes()

loo = skms.LeaveOneOut()
scores = skms.cross_val_score(linreg,
                              diabetes.data, diabetes.target,
                              cv=loo,
                              scoring='neg_mean_squared_error')

scores = pd.Series(np.sqrt(-scores))
df = pd.DataFrame({'RMSE':scores})
df.index.name = 'Repeat'

display(df.describe().T)

ax = sns.swarmplot(y='RMSE', data=df)
ax.set_xlabel('Over LOO\nTrain-Test Splits');
```

	count	mean	std	min	25%	50%	75%	max
RMSE	442.000	44.356	32.197	0.208	18.482	39.547	63.973	158.236



Curiously, there are three noticeable points with a high RMSE and there are about twenty points that form a distinct peak above the main body of the errors (RMSE > 100). That means there are about twenty points that are resistant to prediction with the model we are building using almost all of the data. It would be worthwhile to investigate any common factors in those difficult examples.

LOOCV is a deterministic evaluation method. There's no randomness in the selection because everything is used in the same way every time we run LOOCV. This determinism can be useful for comparing and testing correctness of learning algorithms. However, it can be expensive to run LOOCV because we need to train the model once for each left-out example. Some models have mathematical tricks that can used to drastically reduce the overhead of retraining. On the evaluation side, the net effect of incorporating lots of training data—all but one example—in every CV partition is that LOOCV gives a relatively unbiased estimate of the real error rate. Because the single-example predictions are so closely related—most of the training data is shared and piped into the same learning algorithm—the estimates of our performance error on *new* examples can vary widely. Overall, a general recommendation is to prefer 5- or 10-fold CV to LOOCV.

5.6 Break-It-Down: Deconstructing Error into Bias and Variance

Let's imagine that we are at a race track and we start taking some basic measurements. We see cars zooming around the track and we measure where the cars are at and how fast they are going. Let's say we record two lap times for a total distance $d = 2$. We'll also record an average speed s. If I have a table of these values and I don't remember any high-school physics, I can start trying to relate the different columns together. For example, from the two times and the total time I might come up with the fact that $t_1 + t_2 = t_{\text{total}}$. Again, imagine we *forgot* everything we learned in high-school physics or maybe even fourth-grade math.

Driver	t_1	t_2	t_{total}	s	d
Mario	35	75	110	.018	2
Luigi	20	40	60	.033	2
Yoshi	40	50	90	.022	2

Let's consider some variations in trying to relate the columns. First, are the measurements *perfect* or are there errors in how we recorded them? Second, what relationships—what mathematical operations—am I allowed to use to relate the columns? To keep things under control, we'll limit ourselves to two simple operations, addition and multiplication, and see what happens with them. Lastly, we'll consider relationships between different sets of columns as inputs and outputs.

In Table 5.2, I've laid out the different possibilities and some assessments of how perfectly they can describe the data and what goes wrong.

Table 5.2 Sources of errors in learning.

Inputs	Output	Measurement errors	True Relationship	Try to Relate With	Perfect?	Why?
t_1, t_2	t_{total}	no	add	add	yes	
t_1, t_2	t_{total}	yes	add	add	no	measurement errors
t_{total}, s	d	no	multiply	add	no	can't get right form

Two of these three cases are subpar. The two cases where we end up with "Perfect? No!" are equivalent to two sources of error that we must address when we develop learning systems. A third source of error is the interaction between the training data and the learner. We saw hints of this interaction when we saw the different results from training on different training sets. Together, these three examples of error give us a great foundation to break down the ways we can make mistakes in predictions. Measurement errors—the second line in Table 5.2—reduce our ability to relate values clearly, but those errors may be difficult to control. They may not even be our fault if someone else did the measuring; we're doing the modeling. But the third line, where we have a mismatch between *reality* and our chosen *model*, is a problem of our own making.

5.6.1 Variance of the Data

When we make a mistake—when we have an incorrect class or a MSE greater than zero—there can be a few different causes. One of these causes—the actual randomness in the relationship between the input features and the output target—we have no real control over. For example, not every college graduate that majored in economics and has five years of professional work experience earns the same amount of money. There is a wide range of possibilities for their income. If we include more information, such as selectiveness of their undergrad school, we may be able to narrow down that range. However, randomness will still remain. Similarly, depending on our timing devices and user error at the race track, we may record the times more or less precisely (repeatably).

Having a range of outputs is a fundamental difference between the mathematical functions you saw in high school and random processes. Instead of one input having one-and-only-one output, a single input can have a range—that's a distribution—of outputs. We've wrapped back around to rolling a die or flipping a coin: dealing with randomness. The degree to which our data is affected by randomness—either in measurement or in real-world differences—is called the *variance of the data*.

5.6.2 Variance of the Model

There are some sources of error we *can* control in a learning system, but there may be limits on our control. When we pick a single model—say, linear regression—and go through a training step, we set the values of the parameters of that model. We are setting the values on the knobs of our factory machine. If we choose our training and testing datasets at random, which we should, we lose some control over the outcome. The parameters of our model—the values of our knobs—are subject to the coin-flipping choice of training data. If we flip the coins again, we get different training data. With different training data we get a different trained model. The way models vary due to the random selection of the data we train on is called the *variance of the model*.

A trained model will give us different answers when we use it on test cases and in the wild. Here's a concrete example. If we have one very bad data point, with 1-Nearest Neighbors, most of our training and testing examples will be unaffected by it. However, for anyone that *is* the nearest neighbor of the bad example, things will go wrong. Conversely, if we used a large number of neighbors, the effect of that example would be diluted out among many other training examples. We've ended up with a tradeoff: being able to account for tricky examples also leaves us exposed to following bad examples. Our racetrack example did *not* include an example of variance due to the model training.

5.6.3 Bias of the Model

Our last source of error is where we have the most control. When I choose between two models, one may have a fundamentally better resonance with the relationship between the inputs and outputs. We've already seen one example of poor resonance in Section 5.3.2: a line has great difficulty following the path of a parabola.

Let's tie this idea to our current discussion. We'll start by eliminating noise—the inherent randomness—we discussed a few paragraphs back. We eliminate it by considering only a best-guess output for any given input. So, while an input example made from level of education, degree program, and years post-graduation $\{college, economics, 5\}$ *actually* has a range of possible income predictions, we'll take one best value to represent the possible outputs. Now, we ask, "How well can model one line up with that single value?" and "How well can model two line up with that single value?" Then, we expand that process to *all* of our inputs—$\{secondary, vocational, 10\}$, $\{grad, psychology, 8\}$—and ask how well do the models match the single best guesses for every possible input.

Don't worry, we'll make these ideas more concrete in a moment.

We say that a model that cannot match the *actual* relationship between the inputs and outputs—after we ignore the inherent noisiness in the data—has higher *bias*. A highly biased model has difficulty capturing complicated patterns. A model with low bias can follow more complicated patterns. In the racetrack example, when we wanted to relate speed and time to come up with a distance, we couldn't do it with *addition* because the true relationship between them is *multiplication*.

5.6.4 All Together Now

These three components give us a fundamental breakdown of the sources of errors in our predictions. The three components are (1) the inherent variability in our data, (2) the variability in creating our predicting model from training data, and (3) the bias of our model. The relationship between these and our overall error is called the *bias-variance decomposition*, written mathematically as

$$\text{Error} = \text{Bias}_{\text{Learner}} + \text{Variance}_{\text{Learner(Training)}} + \text{Variance}_{\text{Data}}$$

I'm sweeping many details of this equation under the carpet. But take heart! Even graduate-level, mathematically inclined textbooks sweep details of this particular equation under the carpet. Of course, they call it "removing unnecessary details," but we won't hold it against them. I'll just say that we're in good company. Before we look at some examples, let's reiterate one more time. The errors in our predictions are due to randomness in the data, variability in building our model from training data, and the difference between what relationships our model can express and the actual, *true* relationship.

5.6.5 Examples of Bias-Variance Tradeoffs

Let's examine a few concrete examples of the bias-variance tradeoff by looking at how it applies to *k*-Nearest Neighbors, Linear Regression, and Naive Bayes.

5.6.5.1 Bias-Variance for *k*-NN

Let's think about what happens with *k*-NN as we vary the number of neighbors. Start by going to extremes. The fewest number of neighbors we could use is one. This amounts to saying, "If I'm a new example, then find who is most like me and label me with their target." 1-NN, as a strategy, has the potential to have a very jagged or wiggly border. Every training example gets to have its own say without consulting anyone else! From the opposite perspective, once we find the closest example, we ignore what everyone else says. If there were ten training examples, once we find our closest neighbor, nothing about the other nine matters.

Now, let's go to the opposite extreme. Let's say we have ten examples and we do 10-NN. Our strategy becomes "I'm a new example. Find my ten closest neighbors and average their target. That's my predicted target." Well, with just ten total examples, every new example we come across is going to have exactly those ten nearest neighbors. So, regardless of the example, we are averaging *everyone's* target value. This is equivalent to saying, "Make my predicted target the overall training mean." Our predictions here have no border: they are all exactly the same. We predict the same value regardless of the input predictor values. The only more biased prediction would be predicting some constant—say, 42—that isn't computed from the data at all.

Figure 5.9 summarizes the bias-variance tradeoff for *k*-NN. Increasing the number of neighbors increases our bias and decreases our variance. Decreasing the number of neighbors increases our variance and decreases our bias.

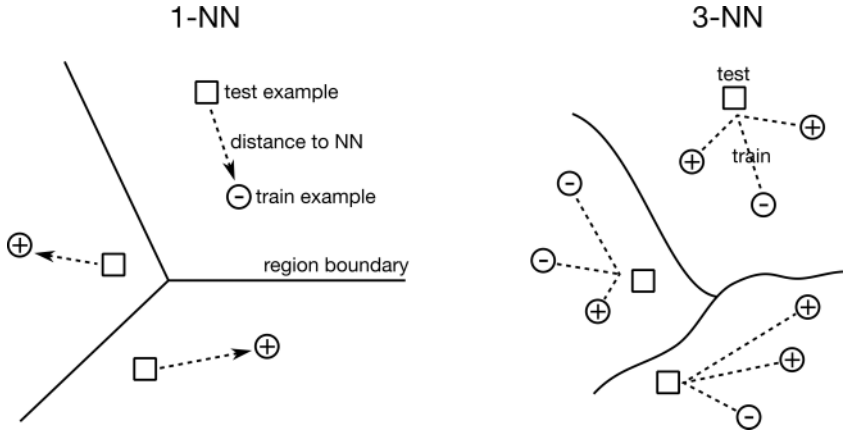


Figure 5.9 Bias in k-NN.

5.6.5.2 Bias-Variance for Linear Regression

What is the comparable analysis for linear regression? There are two different ways to think about it and I want to simplify both of them for now. Let's modify a plain-vanilla linear regression in two ways:

- Restricting the features that are included
- *Adding* new pseudo-features that have a simple relationship to the original features

We'll start with two possible linear regression models. *ConstantLinear* is just predicting a flat horizontal line or surface. The second model, *PlainLinear*, is our standard line or plane-like model that can incline and tilt. In terms of the weights we discussed in Section 4.3.2, the first model sets all weights except w_0 to zero and gives the same output value regardless of input. It says, "I'm afraid of change, don't confuse me with data." The second model says, "It's a party, invite everyone!" You can imagine a middle ground between these two extremes: pick and choose who to invite to the party. That is, set *some* of the weights to zero. So, we have four variations on the linear regression model:

- Constant linear: include no features, $w_i = 0$ for all $i \neq 0$.
- Few: include a few features, most $w_i = 0$.
- Many: include many features, a few $w_i = 0$.
- Plain linear: include all features, no $w_i = 0$.

These give us a similar spectrum of complexities for our linear regression model as we saw with *k*-NN. As we include fewer features by setting more weights to zero, we lose our ability to distinguish between differences represented in the lost features. Put another way, our world gets *flatter* with respect to the missing dimensions. Think about taking a soda can and flattening it like a pancake. Any of the *height* information about the can has been completely lost. Even irregularities in the can—like that little lip that collects spilled out soda—are gone. Shining a light on objects (as in Figure 5.10) gives us an analogy for how differences are hidden when we lose information.

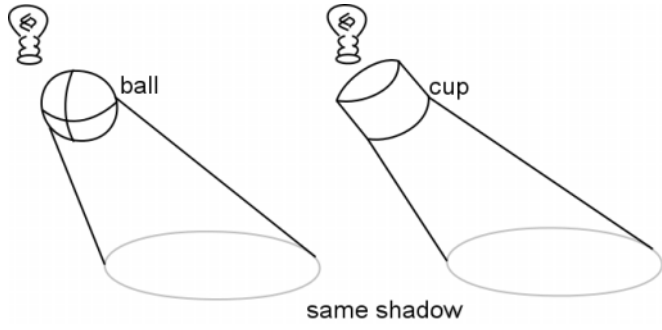


Figure 5.10 Losing features (dimensions) restricts our view of the world and increases our bias.

Knocking out some of the features completely, by setting w_i's to zero, is quite extreme. We'll see a more gradual process of making the weights smaller, but not necessarily zero, in Section 9.1.

Now let's turn to *extending* the features we include. As we saw earlier in this chapter, by adding more *polynomial* terms—x^2, x^3, and friends—we can accommodate more bends or wiggles in our data (Figure 5.11). We can use those bends to capture examples that appear to be oddities. As we also saw, that means we can be fooled by noise.

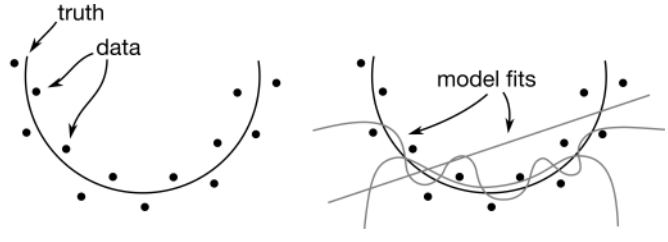


Figure 5.11 Adding complex terms lets our model wiggle more, but that variance might follow noise.

In linear regression, adding features—like polynomial terms—decreases bias but increases variance. Conversely, forcing the weights of features to zero increases bias and decreases variances.

5.6.5.3 Relating *k*-NN and Linear Regression

There is a nice connection between the different linear regression models and the *k*-NN spectrum. The constant linear model—the simplest and most biased model—predicts a single value everywhere: *the mean*. Similarly, a *k*-NN system that has *k* equal to the number of examples in the training dataset takes into account all of the data and summarizes it. That summary can be *the mean*. The most biased linear regression and nearest-neighbors models *both predict the mean*.

On the other end of the spectrum—the end with less bias—we get complexity in two very different ways. *PlainLinear* includes information from all the features, but trades off feature values, based on parameter weights, to get to a "central" predicted value. *1*-NN includes information from all the *features* in the distance calculation, but then it only considers the closest *example* to get the prediction value.

It is amazingly interesting—at least to me—that consulting *more examples* in nearest neighbors leads to *more bias*, yet consulting *more features* in linear regression leads to *less bias*. An explanation is that in nearest neighbors, the only thing we do when we consult someone else is average away the differences between examples—we are smoothing out the rough edges. So, it has as much to do with our method of combining information as it does with the fact that we are consulting more examples.

5.6.5.4 Bias-Variance for Naive Bayes

There's an elephant in the room. We've discussed each of the methods we introduced in Part I, except Naive Bayes (NB). So, what about the bias-variance of NB? Describing the tradeoffs with NB is a little different because Naive Bayes is more like a single point on a spectrum of assumptions. The spectrum that NB lives on has *the number of conditional independence assumptions* on its x axis for complexity. Naive Bayes makes almost as many of these as possible: everything is conditionally independent given the class (Figure 5.12).

If class is independent of all the features—the ultimate independence assumption—the best we can do is guess based on the distribution of the class. For a continuous target, it implies that we guess the mean. We are back to a mean-only model for our least complex, most biased, most-assumptions model within a class of models. That's pretty convenient. Other, more complicated models that try to add complexity to NB could make fewer and fewer independence assumptions. These would come with more and more complicated claims about dependency. Eventually, we get the most complicated type of dependency: the dreaded *full joint distribution*. Among other problems, if we want to adequately capture the distinctions in a fully dependent joint distribution, we need an amount of data that is exponential in the number of features of the data. For each additional feature we need something like—I'm taking liberties with the values—another factor-of-10 examples. If we need 100 examples for two features, we would need 1000 for three features. Yikes. Suffice it to say, that is not good for us.

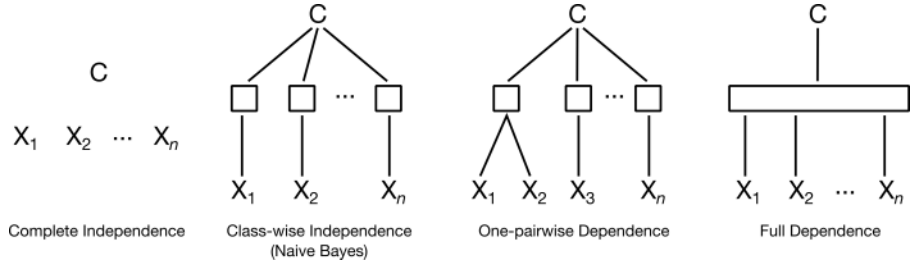


Figure 5.12 Naive Bayes is one bias-variance point on a spectrum of possibilities.

5.6.5.5 Summary Table

It might be surprising that three learning methods, each motivated by different priorities and conceptualizations of the data, all have a common starting point: in a simple enough scenario, they predict the *mean*. Nearest neighbors with every example? It predicts a mean, if that's our summary calculation. Linear regression with only w_0? Yup, that w_0 ends up being the mean! A simpler form of Naive Bayes—Even Naiver Bayes?—is exactly the mean (or the most frequent value for a classification problem) of the output target. Each method extends that in different ways, however. Table 5.3 shows how the models have different tradeoffs in terms of bias-variance and under- and overfitting.

Table 5.3 Tradeoffs between bias and variance.

Scenario	Example	Good	Bad	Risk
high bias & low variance	more neighbors	resists noise	misses pattern	underfit
	low-degree polynomial	forced to generalize		
	smaller or zero linear regression coefficients			
	more independence assumptions			
low bias & high variance	fewer neighbors	follows complex patterns	follows noise	overfit
	high-degree polynomial		memorizes training data	
	bigger linear regression coefficients			
	fewer independence assumptions			

5.7 Graphical Evaluation and Comparison

Our discussion has turned a bit towards the theoretical. You can probably tell because we haven't seen any code for a while. I want to remedy that by going immediately to evaluating performance *visually*. In additional to being fun, evaluating learners visually answers important questions in ways that can be hard to boil down to a single number. Remember, single numbers—mean, median, I'm looking at *you*—can be highly misleading.

5.7.1 Learning Curves: How Much Data Do We Need?

One of the simplest questions we can ask about a learning system is how its learning performance increases as we give it more training examples. If the learner *never* gets good performance, even when we give it a lot of data, we might be barking up the wrong proverbial tree. We could also see that when we use more, or all, of our training data, we continue improving our performance. That might give us confidence to spend some real-world effort to obtain more data to train our model. Here's some code to get us started. `sklearn` provides `learning_curve` to do the calculations we need.

In [33]:

```
iris = datasets.load_iris()

# 10 data set sizes:  10% - 100%
# (that much data is piped to a 5-fold CV)
train_sizes = np.linspace(.1, 1.0, 10)
nn = neighbors.KNeighborsClassifier()

(train_N,
 train_scores,
 test_scores) = skms.learning_curve(nn, iris.data, iris.target,
                                    cv=5, train_sizes=train_sizes)

# collapse across the 5 CV scores; one result for each data set size
df = pd.DataFrame(test_scores, index=(train_sizes*100).astype(np.int))
df['Mean 5-CV'] = df.mean(axis='columns')
df.index.name = "% Data Used"

display(df)
```

% Data Used	0	1	2	3	4	Mean 5-CV
10	0.3333	0.3333	0.3333	0.3333	0.3333	0.3333
20	0.3333	0.3333	0.3333	0.3333	0.3333	0.3333
30	0.3333	0.3333	0.3333	0.3333	0.3333	0.3333
40	0.6667	0.6667	0.6667	0.6667	0.6667	0.6667
50	0.6667	0.6667	0.6667	0.6667	0.6667	0.6667
60	0.6667	0.6667	0.6667	0.6667	0.6667	0.6667
70	0.9000	0.8000	0.8333	0.8667	0.8000	0.8400
80	0.9667	0.9333	0.9000	0.9000	0.9667	0.9333
90	0.9667	1.0000	0.9000	0.9667	1.0000	0.9667
100	0.9667	1.0000	0.9333	0.9667	1.0000	0.9733

`learning_curve` returns arrays with two dimensions: the number of training sizes by the number of cv-folds. I'll call these `(percents, folds)`. In the code above, the values are `(10, 5)`. Unfortunately, turning those values—from the table immediately above—into graphs can be a bit of a headache.

Fortunately, Seaborn has—or had, it's being deprecated—a helper we can use. We're going to send the results to `tsplot`. `tsplot` creates multiple overlaid graphs, one for each condition, and it gives us a center and a range based on the repeated measurements we have. Remember how our college grads might have a range of possible incomes? It's the same idea here. `tsplot` is geared towards plotting time series. It expects data with three components: times, conditions, and repeats. In turn, these three components become the x axis, the grouper (one line), and the repeats (the width around the line). The grouping serves to keep certain data points together; since we're drawing multiple plots on one figure, we need to know which data belongs together in *one* shade. The repeats are the multiple assessments of the same scenario subject to some random variation. Do it again and you get a slightly different result. `tsplot` expects these components in the following order: `(repeats, times, conds)`.

If we take the results of `learning_curve` and stack `train_scores` and `test_scores` on the outermost dimension, we end up with data that is structured like `(train/test condition, percents, folds)`. We just need those dimensions turned inside out since `(folds, percents, conditions)` lines up with `tsplot`'s `(repeats, times, conditions)`. The way we do this is with `np.transpose`.

In [34]:

```
# tsplot expects array data to have these dimensions:
# (repeats, times, conditions)
# for us, those translate to:
# (CV scores, percents, train/test)
joined = np.array([train_scores, test_scores]).transpose()

ax = sns.tsplot(joined,
                time=train_sizes,
                condition=['Train', 'Test'],
                interpolate=False)

ax.set_title("Learning Curve for 5-NN Classifier")
ax.set_xlabel("Number of Samples used for Training")
ax.set_ylabel("Accuracy");
```