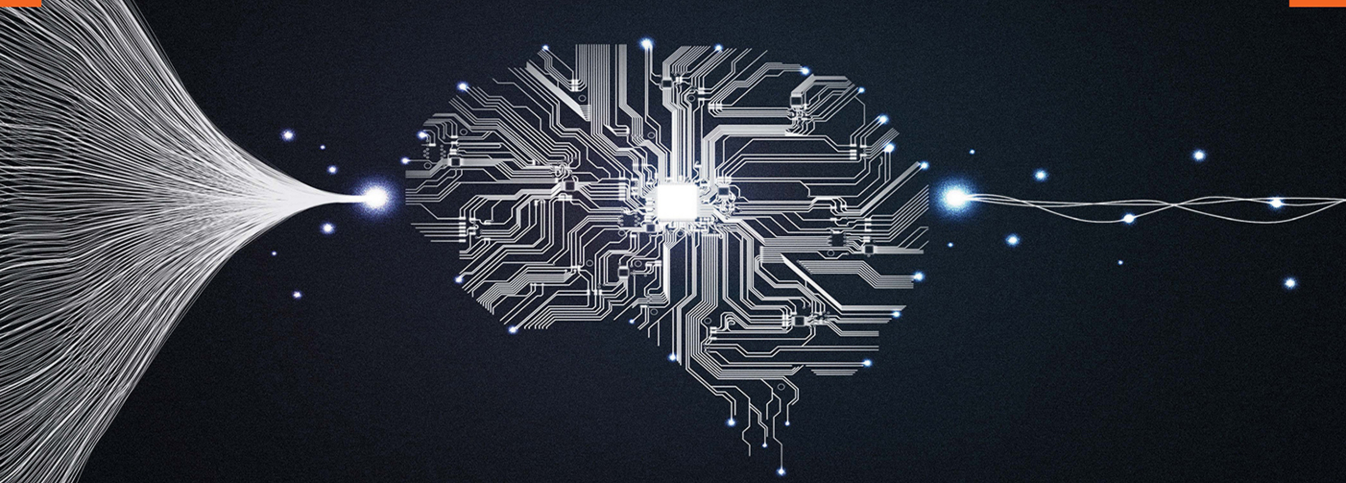
Addison Wesley Data & Analytics Series

Machine Learning with Python for Everyone



Mark E. Fenner

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Machine Learning with Python for Everyone

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

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department
at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2019938761

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.

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)]))
```

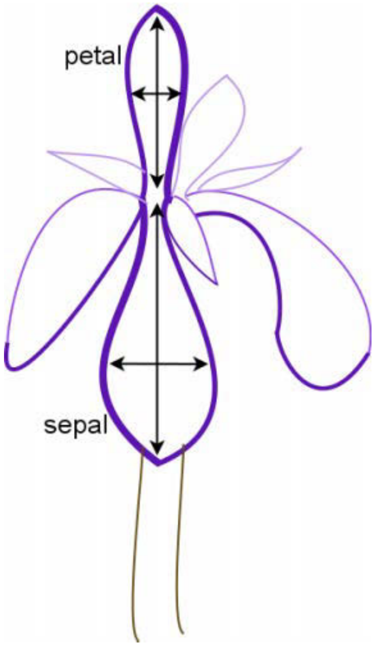
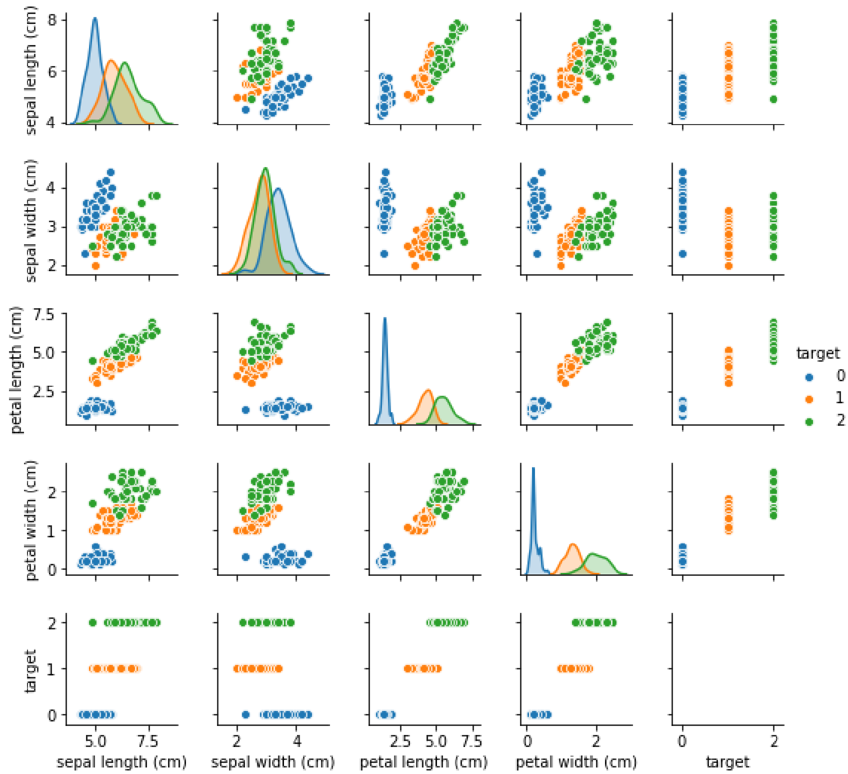


Figure 3.1 An iris and its parts.

	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 *vs*, *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).

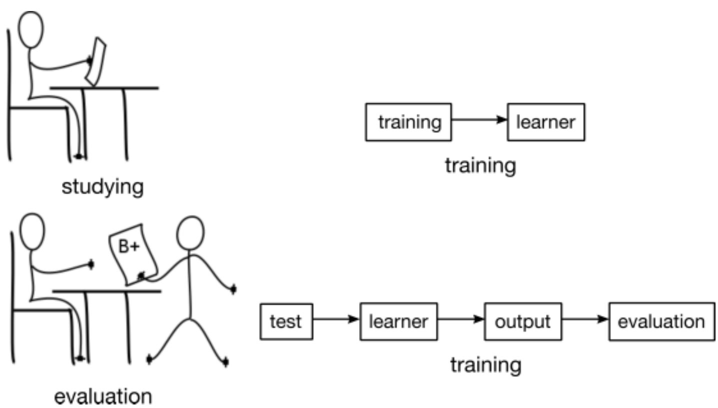


Figure 3.2 School work: training, testing, and evaluating.

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

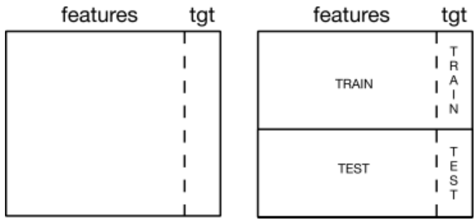


Figure 3.3 Training and testing with features and a target in a table.

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

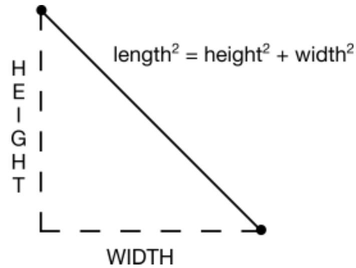


Figure 3.4 Distances from components.

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.

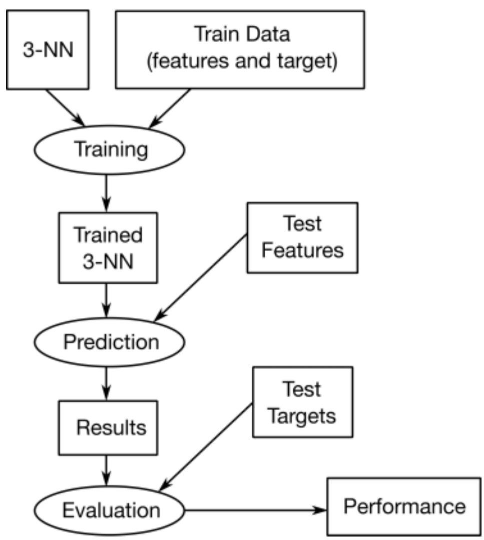


Figure 3.5 Workflow of training, testing, and evaluation for 3-NN.

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.

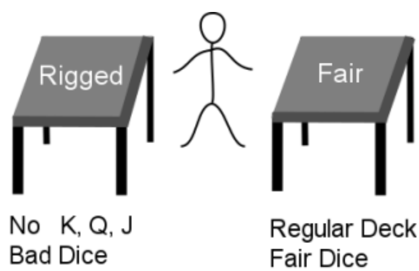


Figure 3.6 Fair and rigged tables at a casino.

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

Index

Symbols

A

B

C

D

G

H

I

M

N

Q

S

T

X

Y

Z