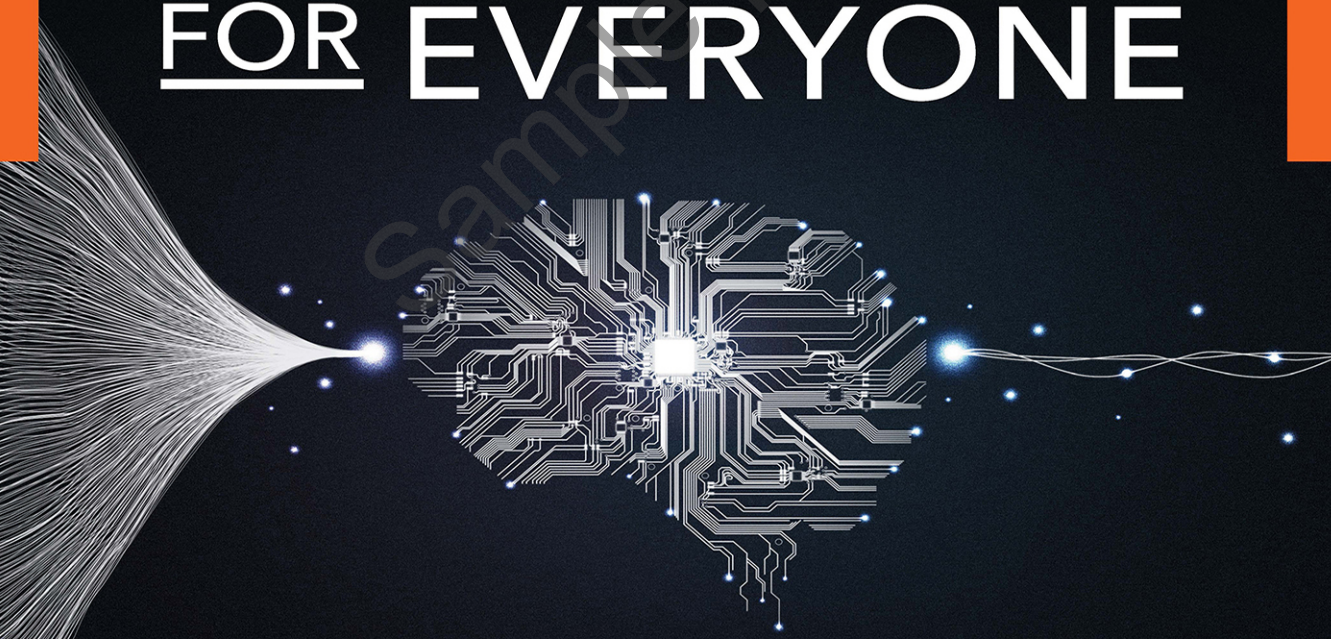
ADDISON WESLEY DATA & ANALYTICS SERIES

MACHINE LEARNING WITH PYTHON FOR EVERYONE



MARK E. FENNER

Contents

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)]))
```

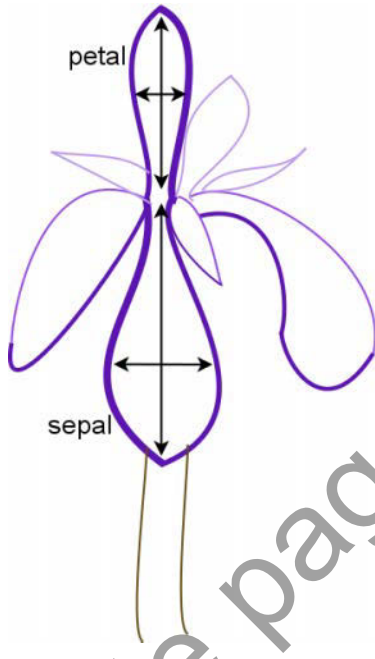
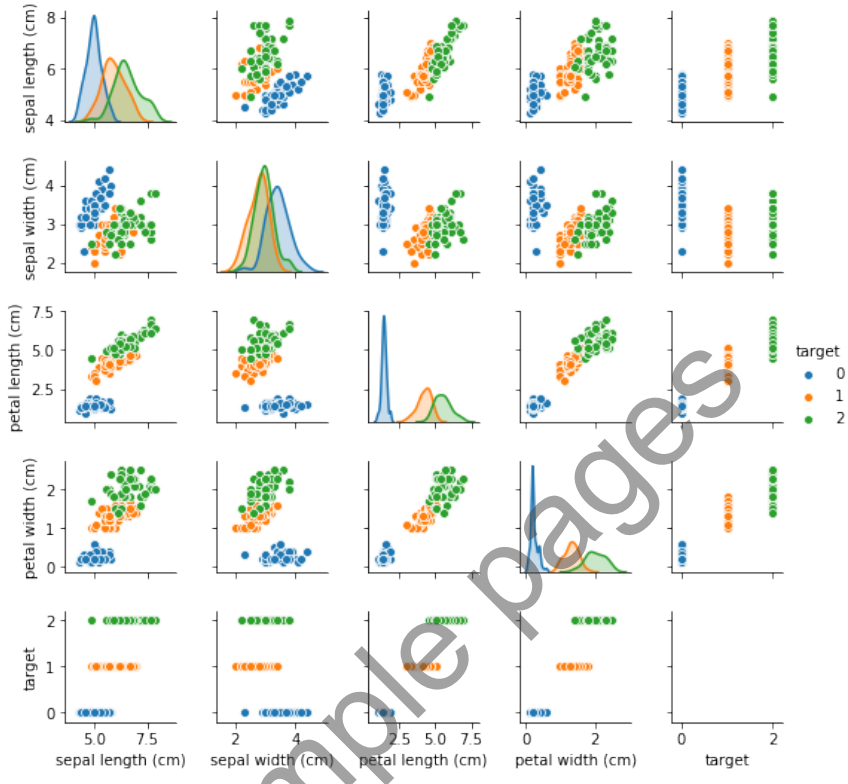


Figure 3.1 An iris and its parts.

	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).

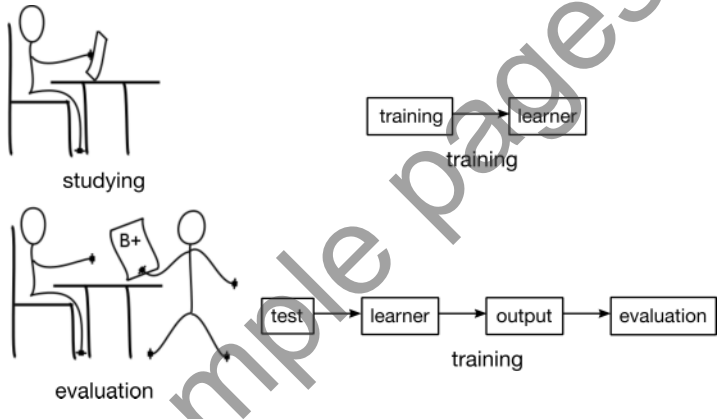


Figure 3.2 School work: training, testing, and evaluating.

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

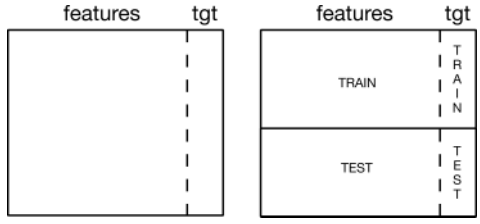


Figure 3.3 Training and testing with features and a target in a table.

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

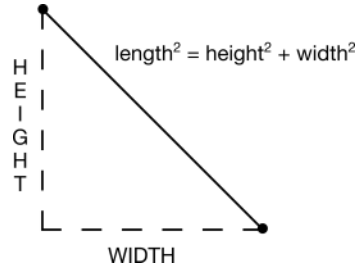


Figure 3.4 Distances from components.

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.

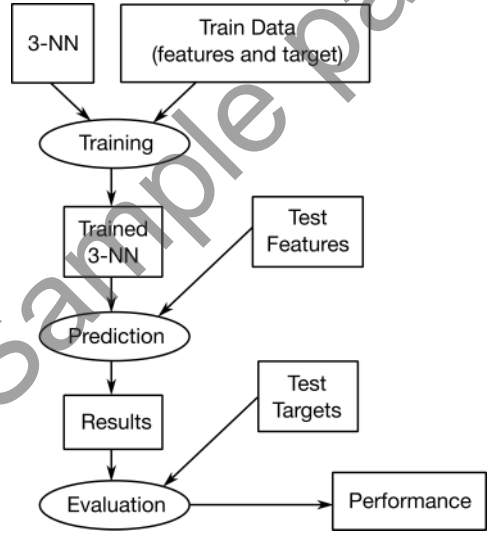


Figure 3.5 Workflow of training, testing, and evaluation for 3-NN.

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.

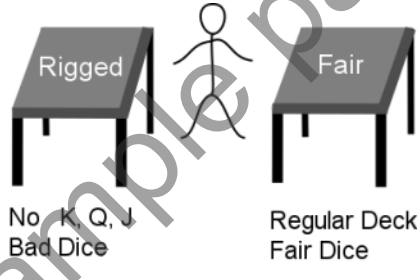


Figure 3.6 Fair and rigged tables at a casino.

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 `imports` 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()
```