3 Machine learning

This chapter covers

- Understanding why data scientists use machine learning
- Identifying the most important Python libraries for machine learning
- Discussing the process for model building
- Using machine learning techniques
- Gaining hands-on experience with machine learning

Do you know how computers learn to protect you from malicious persons? Computers filter out more than 60% of your emails and can learn to do an even better job at protecting you over time.

Can you explicitly teach a computer to recognize persons in a picture? It's possible but impractical to encode all the possible ways to recognize a person, but you'll soon see that the possibilities are nearly endless. To succeed, you'll need to add a new skill to your toolkit, *machine learning*, which is the topic of this chapter.

3.1 What is machine learning and why should you care about it?

> "Machine learning is a field of study that gives computers the ability to learn without being explicitly programmed."
>
> —Arthur Samuel, 1959[1]

The definition of machine learning coined by Arthur Samuel is often quoted and is genius in its broadness, but it leaves you with the question of how the computer learns. To achieve machine learning, experts develop general-purpose algorithms that can be used on large classes of learning problems. When you want to solve a *specific task* you only need to feed the algorithm more *specific data*. In a way, you're programming by example. In most cases a computer will use data as its source of information and compare its output to a desired output and then correct for it. The more data or "experience" the computer gets, the better it becomes at its designated job, like a human does.

When machine learning is seen as a process, the following definition is insightful:

> "Machine learning is the process by which a computer can work more accurately as it collects and learns from the data it is given."
>
> —Mike Roberts[2]

For example, as a user writes more text messages on a phone, the phone learns more about the messages' common vocabulary and can predict (autocomplete) their words faster and more accurately.

In the broader field of science, machine learning is a subfield of artificial intelligence and is closely related to applied mathematics and statistics. All this might sound a bit abstract, but machine learning has many applications in everyday life.

3.1.1 Applications for machine learning in data science

Regression and *classification* are of primary importance to a data scientist. To achieve these goals, one of the main tools a data scientist uses is machine learning. The uses for regression and automatic classification are wide ranging, such as the following:

- Finding oil fields, gold mines, or archeological sites based on existing sites (classification and regression)
- Finding place names or persons in text (classification)
- Identifying people based on pictures or voice recordings (classification)
- Recognizing birds based on their whistle (classification)

---

[1] Although the following paper is often cited as the source of this quote, it's not present in a 1967 reprint of that paper. The authors were unable to verify or find the exact source of this quote. See Arthur L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *IBM Journal of Research and Development* 3, no. 3 (1959):210–229.

[2] Mike Roberts is the technical editor of this book. Thank you, Mike.

- Identifying profitable customers (regression and classification)
- Proactively identifying car parts that are likely to fail (regression)
- Identifying tumors and diseases (classification)
- Predicting the amount of money a person will spend on product X (regression)
- Predicting the number of eruptions of a volcano in a period (regression)
- Predicting your company's yearly revenue (regression)
- Predicting which team will win the Champions League in soccer (classification)

Occasionally data scientists build a *model* (an abstraction of reality) that provides insight to the underlying processes of a phenomenon. When the goal of a model isn't prediction but interpretation, it's called *root cause analysis*. Here are a few examples:

- Understanding and optimizing a business process, such as determining which products add value to a product line
- Discovering what causes diabetes
- Determining the causes of traffic jams

This list of machine learning applications can only be seen as an appetizer because it's ubiquitous within data science. Regression and classification are two important techniques, but the repertoire and the applications don't end, with clustering as one other example of a valuable technique. Machine learning techniques can be used throughout the data science process, as we'll discuss in the next section.

3.1.2 Where machine learning is used in the data science process

Although machine learning is mainly linked to the data-modeling step of the data science process, it can be used at almost every step. To refresh your memory from previous chapters, the data science process is shown in figure 3.1.

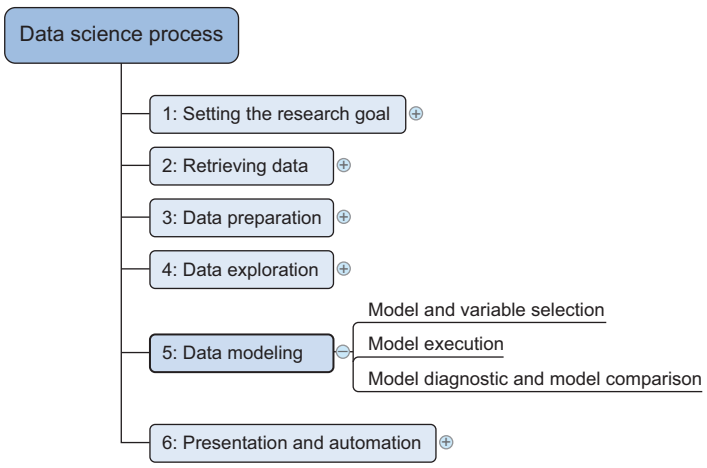


Figure 3.1 The data science process

The data modeling phase can't start until you have qualitative raw data you can understand. But prior to that, the *data preparation* phase can benefit from the use of machine learning. An example would be cleansing a list of text strings; machine learning can group similar strings together so it becomes easier to correct spelling errors.

Machine learning is also useful when *exploring data*. Algorithms can root out underlying patterns in the data where they'd be difficult to find with only charts.

Given that machine learning is useful throughout the data science process, it shouldn't come as a surprise that a considerable number of Python libraries were developed to make your life a bit easier.

3.1.3 Python tools used in machine learning

Python has an overwhelming number of packages that can be used in a machine learning setting. The Python machine learning ecosystem can be divided into three main types of packages, as shown in figure 3.2.

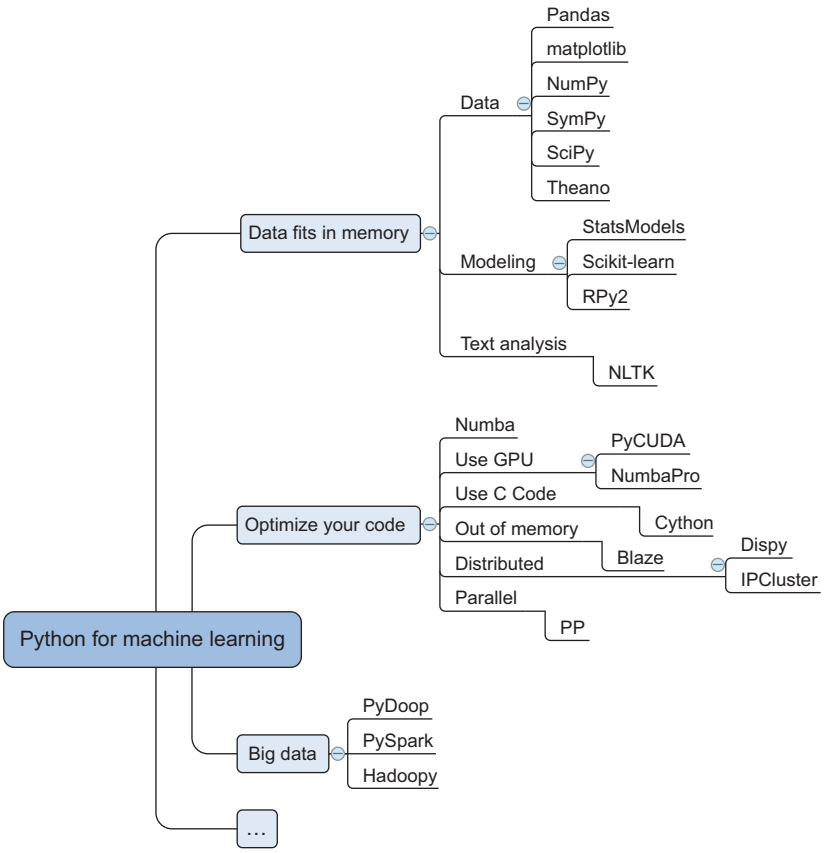


Figure 3.2 Overview of Python packages used during the machine-learning phase

The first type of package shown in figure 3.2 is mainly used in simple tasks and when data fits into memory. The second type is used to optimize your code when you've finished prototyping and run into speed or memory issues. The third type is specific to using Python with big data technologies.

PACKAGES FOR WORKING WITH DATA IN MEMORY

When prototyping, the following packages can get you started by providing advanced functionalities with a few lines of code:

- *SciPy* is a library that integrates fundamental packages often used in scientific computing such as NumPy, matplotlib, Pandas, and SymPy.
- *NumPy* gives you access to powerful array functions and linear algebra functions.
- *Matplotlib* is a popular 2D plotting package with some 3D functionality.
- *Pandas* is a high-performance, but easy-to-use, data-wrangling package. It introduces dataframes to Python, a type of in-memory data table. It's a concept that should sound familiar to regular users of R.
- *SymPy* is a package used for symbolic mathematics and computer algebra.
- *StatsModels* is a package for statistical methods and algorithms.
- *Scikit-learn* is a library filled with machine learning algorithms.
- *RPy2* allows you to call R functions from within Python. R is a popular open source statistics program.
- *NLTK* (Natural Language Toolkit) is a Python toolkit with a focus on text analytics.

These libraries are good to get started with, but once you make the decision to run a certain Python program at frequent intervals, performance comes into play.

OPTIMIZING OPERATIONS

Once your application moves into production, the libraries listed here can help you deliver the speed you need. Sometimes this involves connecting to big data infrastructures such as Hadoop and Spark.

- *Numba and NumbaPro*—These use just-in-time compilation to speed up applications written directly in Python and a few annotations. NumbaPro also allows you to use the power of your graphics processor unit (GPU).
- *PyCUDA*—This allows you to write code that will be executed on the GPU instead of your CPU and is therefore ideal for calculation-heavy applications. It works best with problems that lend themselves to being parallelized and need little input compared to the number of required computing cycles. An example is studying the robustness of your predictions by calculating thousands of different outcomes based on a single start state.
- *Cython, or C for Python*—This brings the C programming language to Python. C is a lower-level language, so the code is closer to what the computer eventually uses (bytecode). The closer code is to bits and bytes, the faster it executes. A computer is also faster when it knows the type of a variable (called *static typing*). Python wasn't designed to do this, and Cython helps you to overcome this shortfall.

- *Blaze*—Blaze gives you data structures that can be bigger than your computer's main memory, enabling you to work with large data sets.
- *Dispy and IPCluster*—These packages allow you to write code that can be distributed over a cluster of computers.
- *PP*—Python is executed as a single process by default. With the help of PP you can parallelize computations on a single machine or over clusters.
- *Pydoop and Hadoopy*—These connect Python to Hadoop, a common big data framework.
- *PySpark*—This connects Python and Spark, an in-memory big data framework.

Now that you've seen an overview of the available libraries, let's look at the modeling process itself.

3.2 The modeling process

The modeling phase consists of four steps:

1. Feature engineering and model selection
2. Training the model
3. Model validation and selection
4. Applying the trained model to unseen data

Before you find a good model, you'll probably iterate among the first three steps.

The last step isn't always present because sometimes the goal isn't prediction but explanation (root cause analysis). For instance, you might want to find out the causes of species' extinctions but not necessarily predict which one is next in line to leave our planet.

It's possible to *chain* or *combine* multiple techniques. When you chain multiple models, the output of the first model becomes an input for the second model. When you combine multiple models, you train them independently and combine their results. This last technique is also known as *ensemble learning*.

A model consists of constructs of information called *features* or *predictors* and a *target* or *response variable*. Your model's goal is to predict the target variable, for example, tomorrow's high temperature. The variables that help you do this and are (usually) known to you are the features or predictor variables such as today's temperature, cloud movements, current wind speed, and so on. The best models are those that accurately represent reality, preferably while staying concise and interpretable. To achieve this, feature engineering is the most important and arguably most interesting part of modeling. For example, an important feature in a model that tried to explain the extinction of large land animals in the last 60,000 years in Australia turned out to be the population number and spread of humans.

3.2.1 Engineering features and selecting a model

With engineering features, you must come up with and create possible predictors for the model. This is one of the most important steps in the process because a model

recombines these features to achieve its predictions. Often you may need to consult an expert or the appropriate literature to come up with meaningful features.

Certain features are the variables you get from a data set, as is the case with the provided data sets in our exercises and in most school exercises. In practice you'll need to find the features yourself, which may be scattered among different data sets. In several projects we had to bring together more than 20 different data sources before we had the raw data we required. Often you'll need to apply a transformation to an input before it becomes a good predictor or to combine multiple inputs. An example of combining multiple inputs would be *interaction variables*: the impact of either single variable is low, but if both are present their impact becomes immense. This is especially true in chemical and medical environments. For example, although vinegar and bleach are fairly harmless common household products by themselves, mixing them results in poisonous chlorine gas, a gas that killed thousands during World War I.

In medicine, clinical pharmacy is a discipline dedicated to researching the effect of the interaction of medicines. This is an important job, and it doesn't even have to involve two medicines to produce potentially dangerous results. For example, mixing an antifungal medicine such as Sporanox with grapefruit has serious side effects.

Sometimes you have to use modeling techniques to derive features: the output of a model becomes part of another model. This isn't uncommon, especially in text mining. Documents can first be annotated to classify the content into categories, or you can count the number of geographic places or persons in the text. This counting is often more difficult than it sounds; models are first applied to recognize certain words as a person or a place. All this new information is then poured into the model you want to build. One of the biggest mistakes in model construction is the *availability bias*: your features are only the ones that you could easily get your hands on and your model consequently represents this one-sided "truth." Models suffering from availability bias often fail when they're validated because it becomes clear that they're not a valid representation of the truth.

In World War II, after bombing runs on German territory, many of the English planes came back with bullet holes in the wings, around the nose, and near the tail of the plane. Almost none of them had bullet holes in the cockpit, tail rudder, or engine block, so engineering decided extra armor plating should be added to the wings. This looked like a sound idea until a mathematician by the name of Abraham Wald explained the obviousness of their mistake: they only took into account the planes that returned. The bullet holes on the wings were actually the least of their concern, because at least a plane with this kind of damage could make it back home for repairs. Plane fortification was hence increased on the spots that were unscathed on returning planes. The initial reasoning suffered from availability bias: the engineers ignored an important part of the data because it was harder to obtain. In this case they were lucky, because the reasoning could be reversed to get the intended result without getting the data from the crashed planes.

When the initial features are created, a model can be *trained* to the data.

3.2.2 Training your model

With the right predictors in place and a modeling technique in mind, you can progress to model training. In this phase you present to your model data from which it can learn.

The most common modeling techniques have industry-ready implementations in almost every programming language, including Python. These enable you to train your models by executing a few lines of code. For more state-of-the art data science techniques, you'll probably end up doing heavy mathematical calculations and implementing them with modern computer science techniques.

Once a model is trained, it's time to test whether it can be extrapolated to reality: model validation.

3.2.3 Validating a model

Data science has many modeling techniques, and the question is which one is the right one to use. A good model has two properties: it has good predictive power and it generalizes well to data it hasn't seen. To achieve this you define an error measure (how wrong the model is) and a validation strategy.

Two common *error measures* in machine learning are the *classification error rate* for classification problems and the *mean squared error* for regression problems. The classification error rate is the percentage of observations in the test data set that your model mislabeled; lower is better. The mean squared error measures how big the average error of your prediction is. Squaring the average error has two consequences: you can't cancel out a wrong prediction in one direction with a faulty prediction in the other direction. For example, overestimating future turnover for next month by 5,000 doesn't cancel out underestimating it by 5,000 for the following month. As a second consequence of squaring, bigger errors get even more weight than they otherwise would. Small errors remain small or can even shrink (if <1), whereas big errors are enlarged and will definitely draw your attention.

Many *validation strategies* exist, including the following common ones:

- *Dividing your data into a training set with X% of the observations and keeping the rest as a holdout data set* (a data set that's never used for model creation)—This is the most common technique.
- *K-folds cross validation*—This strategy divides the data set into k parts and uses each part one time as a test data set while using the others as a training data set. This has the advantage that you use all the data available in the data set.
- *Leave-1 out*—This approach is the same as k-folds but with k=1. You always leave one observation out and train on the rest of the data. This is used only on small data sets, so it's more valuable to people evaluating laboratory experiments than to big data analysts.

Another popular term in machine learning is *regularization*. When applying regularization, you incur a penalty for every extra variable used to construct the model. With *L1*

regularization you ask for a model with as few predictors as possible. This is important for the model's robustness: simple solutions tend to hold true in more situations. *L2 regularization* aims to keep the variance between the coefficients of the predictors as small as possible. Overlapping variance between predictors makes it hard to make out the actual impact of each predictor. Keeping their variance from overlapping will increase interpretability. To keep it simple: regularization is mainly used to stop a model from using too many features and thus prevent over-fitting.

Validation is extremely important because it determines whether your model works in real-life conditions. To put it bluntly, it's whether your model is worth a dime. Even so, every now and then people send in papers to respected scientific journals (and sometimes even succeed at publishing them) with faulty validation. The result of this is they get rejected or need to retract the paper because everything is wrong. Situations like this are bad for your mental health so always keep this in mind: test your models on data the constructed model has never seen and make sure this data is a true representation of what it would encounter when applied on fresh observations by other people. For classification models, instruments like the confusion matrix (introduced in chapter 2 but thoroughly explained later in this chapter) are golden; embrace them.

Once you've constructed a good model, you can (optionally) use it to predict the future.

3.2.4 Predicting new observations

If you've implemented the first three steps successfully, you now have a performant model that generalizes to unseen data. The process of applying your model to new data is called model scoring. In fact, model scoring is something you implicitly did during validation, only now you don't know the correct outcome. By now you should trust your model enough to use it for real.

Model scoring involves two steps. First, you prepare a data set that has features exactly as defined by your model. This boils down to repeating the data preparation you did in step one of the modeling process but for a new data set. Then you apply the model on this new data set, and this results in a prediction.

Now let's look at the different types of machine learning techniques: a different problem requires a different approach.

3.3 Types of machine learning

Broadly speaking, we can divide the different approaches to machine learning by the amount of human effort that's required to coordinate them and how they use *labeled data*—data with a category or a real-value number assigned to it that represents the outcome of previous observations.

- *Supervised learning* techniques attempt to discern results and learn by trying to find patterns in a labeled data set. Human interaction is required to label the data.
- *Unsupervised learning* techniques don't rely on labeled data and attempt to find patterns in a data set without human interaction.

- *Semi-supervised learning* techniques need labeled data, and therefore human interaction, to find patterns in the data set, but they can still progress toward a result and learn even if passed unlabeled data as well.

In this section, we'll look at all three approaches, see what tasks each is more appropriate for, and use one or two of the Python libraries mentioned earlier to give you a feel for the code and solve a task. In each of these examples, we'll work with a downloadable data set that has already been cleaned, so we'll skip straight to the data modeling step of the data science process, as discussed earlier in this chapter.

3.3.1 Supervised learning

As stated before, supervised learning is a learning technique that can only be applied on labeled data. An example implementation of this would be discerning digits from images. Let's dive into a case study on number recognition.

CASE STUDY: DISCERNING DIGITS FROM IMAGES

One of the many common approaches on the web to stopping computers from hacking into user accounts is the Captcha check—a picture of text and numbers that the human user must decipher and enter into a form field before sending the form back to the web server. Something like figure 3.3 should look familiar.

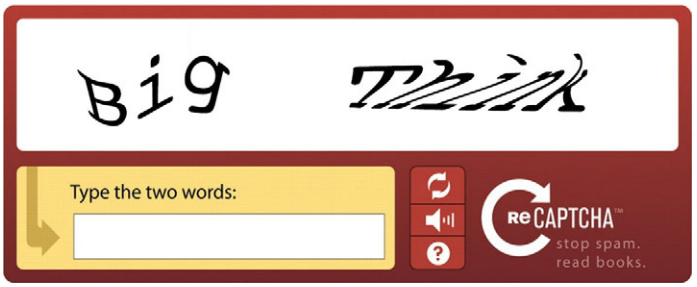


Figure 3.3 A simple Captcha control can be used to prevent automated spam being sent through an online web form.

With the help of the *Naïve Bayes classifier*, a simple yet powerful algorithm to categorize observations into classes that's explained in more detail in the sidebar, you can recognize digits from textual images. These images aren't unlike the Captcha checks many websites have in place to make sure you're not a computer trying to hack into the user accounts. Let's see how hard it is to let a computer recognize images of numbers.

Our research goal is to let a computer recognize images of numbers (step one of the data science process).

The data we'll be working on is the MNIST data set, which is often used in the data science literature for teaching and benchmarking.

Introducing Naïve Bayes classifiers in the context of a spam filter

Not every email you receive has honest intentions. Your inbox can contain unsolicited commercial or bulk emails, a.k.a. spam. Not only is spam annoying, it's often used in scams and as a carrier for viruses. Kaspersky[3] estimates that more than 60% of the emails in the world are spam. To protect users from spam, most email clients run a program in the background that classifies emails as either spam or safe.

A popular technique in spam filtering is employing a classifier that uses the words inside the mail as predictors. It outputs the chance that a specific email is spam given the words it's composed of (in mathematical terms, P(spam | words)). To reach this conclusion it uses three calculations:

- P(spam)—The average rate of spam without knowledge of the words. According to Kaspersky, an email is spam 60% of the time.
- P(words)—How often this word combination is used regardless of spam.
- P(words | spam)—How often these words are seen when a training mail was labeled as spam.

To determine the chance that a new email is spam, you'd use the following formula:

P(spam|words) = P(spam)P(words|spam) / P(words)

This is an application of the rule P(B|A) = P(B) P(A|B) / P(A), which is known as Bayes's rule and which lends its name to this classifier. The "naïve" part comes from the classifier's assumption that the presence of one feature doesn't tell you anything about another feature (feature independence, also called absence of multicollinearity). In reality, features are often related, especially in text. For example the word "buy" will often be followed by "now." Despite the unrealistic assumption, the naïve classifier works surprisingly well in practice.

With the bit of theory in the sidebar, you're ready to perform the modeling itself. Make sure to run all the upcoming code in the same scope because each piece requires the one before it. An IPython file can be downloaded for this chapter from the Manning download page of this book.

The MNIST images can be found in the data sets package of Scikit-learn and are already normalized for you (all scaled to the same size: 64x64 pixels), so we won't need much data preparation (step three of the data science process). But let's first fetch our data as step two of the data science process, with the following listing.

Listing 3.1 Step 2 of the data science process: fetching the digital image data

```
from sklearn.datasets import load_digits
import pylab as pl
digits = load_digits()
```

Imports digits database.

Loads digits.

[3] Kaspersky 2014 Quarterly Spam Statistics Report, http://usa.kaspersky.com/internet-security-center/threats/spam-statistics-report-q1-2014#.VVym9blViko.

Working with images isn't much different from working with other data sets. In the case of a gray image, you put a value in every matrix entry that depicts the gray value to be shown. The following code demonstrates this process and is step four of the data science process: data exploration.

Listing 3.2 Step 4 of the data science process: using Scikit-learn

```
pl.gray()
pl.matshow(digits.images[0])
pl.show()
digits.images[0]
```

Turns image into gray-scale values.

Shows first images.

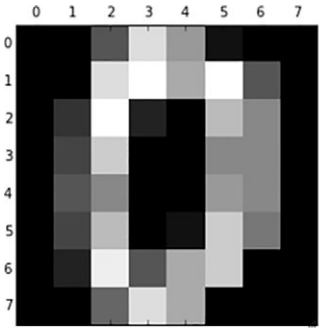
Shows the corresponding matrix.

Figure 3.4 shows how a blurry "0" image translates into a data matrix.

Figure 3.4 shows the actual code output, but perhaps figure 3.5 can clarify this slightly, because it shows how each element in the vector is a piece of the image.

Easy so far, isn't it? There is, naturally, a little more work to do. The Naïve Bayes classifier is expecting a list of values, but `pl.matshow()` returns a two-dimensional array (a matrix) reflecting the shape of the image. To flatten it into a list, we need to call `reshape()` on `digits.images`. The net result will be a one-dimensional array that looks something like this:

```
array([[ 0., 0., 5., 13., 9., 1., 0., 0., 0., 0., 13., 15., 10., 15., 5., 0.,
0., 3., 15., 2., 0., 11., 8., 0., 0., 4., 12., 0., 0., 8., 8., 0.,
0., 5., 8., 0., 0., 9., 8., 0., 0., 4., 11., 0., 1., 12., 7., 0.,
0., 2., 14., 5., 10., 12., 0., 0., 0., 0., 6., 13., 10., 0., 0., 0.]])
```




```
rray([[  0.,   0.,   5.,  13.,   9.,   1.,   0.,   0.],
       [  0.,   0.,  13.,  15.,  10.,  15.,   5.,   0.],
       [  0.,   3.,  15.,   2.,   0.,  11.,   8.,   0.],
       [  0.,   4.,  12.,   0.,   0.,   8.,   8.,   0.],
       [  0.,   5.,   8.,   0.,   0.,   9.,   8.,   0.],
       [  0.,   4.,  11.,   0.,   1.,  12.,   7.,   0.],
       [  0.,   2.,  14.,   5.,  10.,  12.,   0.,   0.],
       [  0.,   0.,   6.,  13.,  10.,   0.,   0.,   0.]])
```

Figure 3.4 Blurry grayscale representation of the number 0 with its corresponding matrix. The higher the number, the closer it is to white; the lower the number, the closer it is to black.

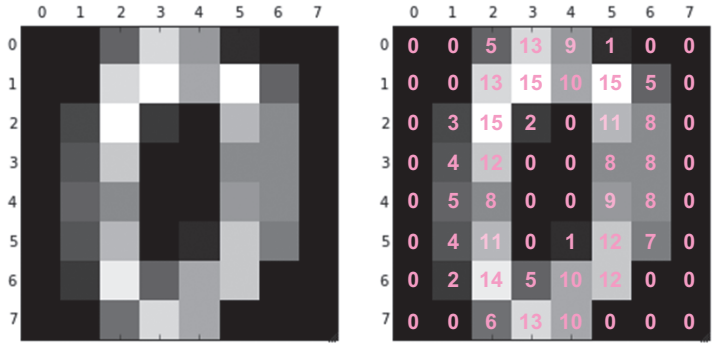


Figure 3.5 We'll turn an image into something usable by the Naïve Bayes classifier by getting the grayscale value for each of its pixels (shown on the right) and putting those values in a list.

The previous code snippet shows the matrix of figure 3.5 flattened (the number of dimensions was reduced from two to one) to a Python list. From this point on, it's a standard classification problem, which brings us to step five of the data science process: model building.

Now that we have a way to pass the contents of an image into the classifier, we need to pass it a training data set so it can start learning how to predict the numbers in the images. We mentioned earlier that Scikit-learn contains a subset of the MNIST database (1,800 images), so we'll use that. Each image is also labeled with the number it actually shows. This will build a probabilistic model in memory of the most likely digit shown in an image given its grayscale values.

Once the program has gone through the training set and built the model, we can then pass it the test set of data to see how well it has learned to interpret the images using the model.

The following listing shows how to implement these steps in code.

Listing 3.3 Image data classification problem on images of digits

```
from sklearn.cross_validation import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import confusion_matrix
import pylab as plt

y = digits.target

n_samples = len(digits.images)
X= digits.images.reshape((n_samples, -1))

print X

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Step 1: Select target variable.

Step 2: Prepare data. Reshape adapts the matrix form. This method could, for instance, turn a 10x10 matrix into 100 vectors.

Step 3: Split into test set and training set.

```
gnb = GaussianNB()
fit = gnb.fit(X_train,y_train)
predicted = fit.predict(X_test)
confusion_matrix(y_test, predicted)
```

Step 5: Fit data.

Step 4: Select a Naïve Bayes classifier; use a Gaussian distribution to estimate probability.

Step 6: Predict data for unseen data.

Step 7: Create confusion matrix.

The end result of this code is called a *confusion matrix*, such as the one shown in figure 3.6. Returned as a two-dimensional array, it shows how often the number predicted was the correct number on the main diagonal and also in the matrix entry (i,j), where j was predicted but the image showed i. Looking at figure 3.6 we can see that the model predicted the number 2 correctly 17 times (at coordinates 3,3), but also that the model predicted the number 8 15 times when it was actually the number 2 in the image (at 9,3).

```
array([[37,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0, 39,  0,  0,  0,  0,  1,  0,  3,  0],
       [ 0,  9, 17,  3,  0,  0,  0,  0, 15,  0],
       [ 0,  0,  0, 38,  0,  0,  0,  2,  5,  0],
       [ 0,  1,  0,  0, 27,  0,  2,  8,  0,  0],
       [ 0,  1,  0,  1,  0, 43,  0,  3,  0,  0],
       [ 0,  0,  0,  0,  0,  0, 52,  0,  0,  0],
       [ 0,  0,  0,  0,  1,  0,  0, 47,  0,  0],
       [ 0,  5,  0,  1,  0,  1,  0,  4, 37,  0],
       [ 0,  2,  0,  7,  1,  0,  0,  3,  7, 27]])
```

Figure 3.6 Confusion matrix produced by predicting what number is depicted by a blurry image

Confusion matrices

A confusion matrix is a matrix showing how wrongly (or correctly) a model predicted, how much it got "confused." In its simplest form it will be a 2x2 table for models that try to classify observations as being A or B. Let's say we have a classification model that predicts whether somebody will buy our newest product: deep-fried cherry pudding. We can either predict: "Yes, this person will buy" or "No, this customer won't buy." Once we make our prediction for 100 people we can compare this to their actual behavior, showing us how many times we got it right. An example is shown in table 3.1.

Table 3.1 Confusion matrix example

Confusion matrix	Predicted "Person will buy"	Predicted "Person will not buy"
Person **bought** the deep-fried cherry pudding	35 (true positive)	10 (false negative)
Person **didn't buy** the deep-fried cherry pudding	15 (false positive)	40 (true negative)

The model was correct in (35+40) 75 cases and incorrect in (15+10) 25 cases, resulting in a (75 correct/100 total observations) 75% accuracy.

All the correctly classified observations are added up on the diagonal (35+40) while everything else (15+10) is incorrectly classified. When the model only predicts two classes (binary), our correct guesses are two groups: true positives (predicted to buy and did so) and true negatives (predicted they wouldn't buy and they didn't). Our incorrect guesses are divided into two groups: false positives (predicted they would buy but they didn't) and false negatives (predicted not to buy but they did). The matrix is useful to see where the model is having the most problems. In this case we tend to be overconfident in our product and classify customers as future buyers too easily (false positive).

From the confusion matrix, we can deduce that for most images the predictions are quite accurate. In a good model you'd expect the sum of the numbers on the main diagonal of the matrix (also known as the matrix *trace*) to be very high compared to the sum of all matrix entries, indicating that the predictions were correct for the most part.

Let's assume we want to show off our results in a more easily understandable way or we want to inspect several of the images and the predictions our program has made: we can use the following code to display one next to the other. Then we can see where the program has gone wrong and needs a little more training. If we're satisfied with the results, the model building ends here and we arrive at step six: presenting the results.

Listing 3.4 Inspecting predictions vs actual numbers

```
images_and_predictions = list(zip(digits.images, fit.predict(X)))
for index, (image, prediction) in enumerate(images_and_predictions[:6]):
    plt.subplot(6, 3 ,index + 5)
    plt.axis('off')
    plt.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    plt.title('Prediction: %i' % prediction)
plt.show()
```

Adds an extra subplot on a 6x3 plot grid. This code could be simplified as: plt.subplot (3, 2 ,index) but this looks visually more appealing.

Stores number image matrix and its prediction (as a number) together in array.

Loops through first 7 images.

Doesn't show an axis.

Shows image in grayscale.

Shows the predicted value as the title to the shown image.

Shows the full plot that is now populated with 6 subplots.

Figure 3.7 shows how all predictions seem to be correct except for the digit number 2, which it labels as 8. We should forgive this mistake as this 2 does share visual similarities

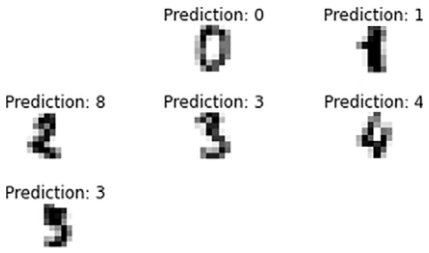


Figure 3.7 For each blurry image a number is predicted; only the number 2 is misinterpreted as 8. Then an ambiguous number is predicted to be 3 but it could as well be 5; even to human eyes this isn't clear.

with 8. The bottom left number is ambiguous, even to humans; is it a 5 or a 3? It's debatable, but the algorithm thinks it's a 3.

By discerning which images were misinterpreted, we can train the model further by labeling them with the correct number they display and feeding them back into the model as a new training set (step 5 of the data science process). This will make the model more accurate, so the cycle of learn, predict, correct continues and the predictions become more accurate. This is a controlled data set we're using for the example. All the examples are the same size and they are all in 16 shades of gray. Expand that up to the variable size images of variable length strings of variable shades of alphanumeric characters shown in the Captcha control, and you can appreciate why a model accurate enough to predict any Captcha image doesn't exist yet.

In this supervised learning example, it's apparent that without the labels associated with each image telling the program what number that image shows, a model cannot be built and predictions cannot be made. By contrast, an unsupervised learning approach doesn't need its data to be labeled and can be used to give structure to an unstructured data set.

3.3.2 Unsupervised learning

It's generally true that most large data sets don't have labels on their data, so unless you sort through it all and give it labels, the supervised learning approach to data won't work. Instead, we must take the approach that will work with this data because

- We can study the *distribution of the data* and infer truths about the data in different parts of the distribution.
- We can study the *structure and values in the data* and infer new, more meaningful data and structure from it.

Many techniques exist for each of these *unsupervised learning* approaches. However, in the real world you're always working toward the research goal defined in the first phase of the data science process, so you may need to combine or try different techniques before either a data set can be labeled, enabling supervised learning techniques, perhaps, or even the goal itself is achieved.

Discerning a simplified latent structure from your data

Not everything can be measured. When you meet someone for the first time you might try to guess whether they like you based on their behavior and how they respond. But what if they've had a bad day up until now? Maybe their cat got run over or they're still down from attending a funeral the week before? The point is that certain variables can be immediately available while others can only be inferred and are therefore missing from your data set. The first type of variables are known as *observable variables* and the second type are known as *latent variables*. In our example, the emotional state of your new friend is a latent variable. It definitely influences their judgment of you but its value isn't clear.

Deriving or inferring latent variables and their values based on the actual contents of a data set is a valuable skill to have because

- Latent variables can substitute for several existing variables already in the data set.
- By reducing the number of variables in the data set, the data set becomes more manageable, any further algorithms run on it work faster, and predictions may become more accurate.
- Because latent variables are designed or targeted toward the defined research goal, you lose little key information by using them.

If we can reduce a data set from 14 observable variables per line to 5 or 6 latent variables, for example, we have a better chance of reaching our research goal because of the data set's simplified structure. As you'll see from the example below, it's not a case of reducing the existing data set to as few latent variables as possible. You'll need to find the sweet spot where the number of latent variables derived returns the most value. Let's put this into practice with a small case study.

Case study: finding latent variables in a wine quality data set

In this short case study, you'll use a technique known as Principal Component Analysis (PCA) to find latent variables in a data set that describes the quality of wine. Then you'll compare how well a set of latent variables works in predicting the quality of wine against the original observable set. You'll learn

1. How to identify and derive those latent variables.
2. How to analyze where the sweet spot is—how many new variables return the most utility—by generating and interpreting a *scree plot* generated by PCA. (We'll look at scree plots in a moment.)

Let's look at the main components of this example.

- *Data set*—The University of California, Irvine (UCI) has an online repository of 325 data sets for machine learning exercises at http://archive.ics.uci.edu/ml/. We'll use the Wine Quality Data Set for red wines created by P. Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis[4]. It's 1,600 lines long and has 11 variables per line, as shown in table 3.2.

[4] You can find full details of the Wine Quality Data Set at https://archive.ics.uci.edu/ml/datasets/Wine+Quality.

Table 3.2 The first three rows of the Red Wine Quality Data Set

Fixed acidity	Volatile acidity	Citric acid	Residual sugar	Chlorides	Free sulfur dioxide	Total sulfur dioxide	Density	pH	Sulfates	Alcohol	Quality
7.4	0.7	0	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5
7.8	0.88	0	2.6	0.098	25	67	0.9968	3.2	0.68	9.8	5
7.8	0.76	0.04	2.3	0.092	15	54	0.997	3.26	0.65	9.8	5

- *Principal Component Analysis*—A technique to find the latent variables in your data set while retaining as much information as possible.
- *Scikit-learn*—We use this library because it already implements PCA for us and is a way to generate the scree plot.

Part one of the data science process is to set our research goal: We want to explain the subjective "wine quality" feedback using the different wine properties.

Our first job then is to download the data set (step two: acquiring data), as shown in the following listing, and prepare it for analysis (step three: data preparation). Then we can run the PCA algorithm and view the results to look at our options.

Listing 3.5 Data acquisition and variable standardization

X is a matrix of predictor variables. These variables are wine properties such as density and alcohol presence.

```
import pandas as pd
from sklearn import preprocessing
from sklearn.decomposition import PCA
import pylab as plt
from sklearn import preprocessing

url = http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/
     winequality-red.csv
data = pd.read_csv(url, sep= ";")
X = data[[u'fixed acidity', u'volatile acidity', u'citric acid',
      u'residual sugar', u'chlorides', u'free sulfur dioxide',
      u'total sulfur dioxide', u'density', u'pH', u'sulphates',
      u'alcohol']]
y = data.quality
X= preprocessing.StandardScaler().fit(X).transform(X)
```

Downloads location of wine-quality data set.

Reads in the CSV data. It's separated by a semi-colon.

y is a vector and represents the dependent variable (target variable). y is the perceived wine quality.

When standardizing data, the following formula is applied to every data point: $z = (x-\mu)/\sigma$, where z is the new observation value, x the old one, μ is the mean, and σ the standard deviation. The PCA of a data matrix is easier to interpret when the columns have first been centered by their means.

With the initial data preparation behind you, you can execute the PCA. The resulting scree plot (which will be explained shortly) is shown in figure 3.8. Because PCA is an explorative technique, we now arrive at step four of the data science process: data exploration, as shown in the following listing.

Listing 3.6 Executing the principal component analysis

```
model = PCA()
results = model.fit(X)
Z = results.transform(X)
plt.plot(results.explained_variance_)
plt.show()
```

Creates instance of principal component analysis class

Applies PCA on predictor variables to see if they can be compacted into fewer variables

Turns result into array so we can use newly created data

Plots explained variance in variables; this plot is a scree plot

Shows plot

Now let's look at the scree plot in figure 3.8.

The plot generated from the wine data set is shown in figure 3.8. What you hope to see is an elbow or hockey stick shape in the plot. This indicates that a few variables can represent the majority of the information in the data set while the rest only add a little more. In our plot, PCA tells us that reducing the set down to one variable can capture approximately 28% of the total information in the set (the plot is zero-based, so variable

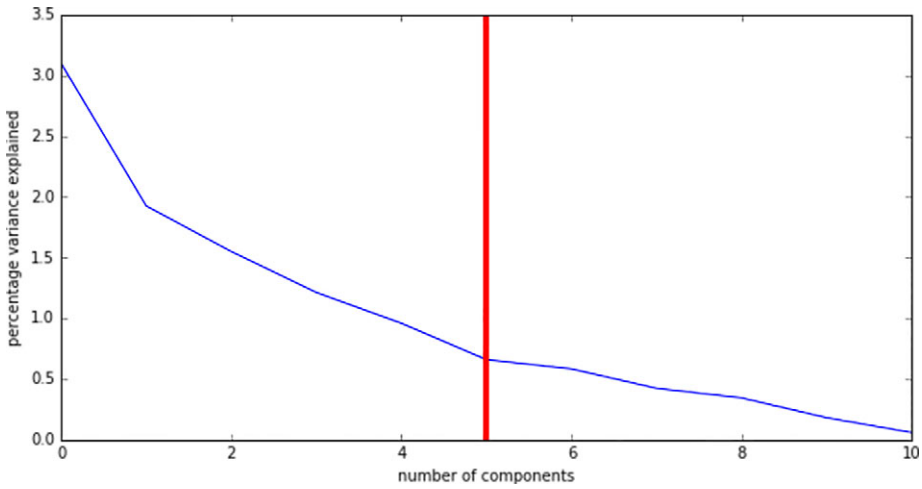


Figure 3.8 PCA scree plot showing the marginal amount of information of every new variable PCA can create. The first variables explain approximately 28% of the variance in the data, the second variable accounts for another 17%, the third approximately 15%, and so on.

one is at position zero on the x axis), two variables will capture approximately 17% more or 45% total, and so on. Table 3.3 shows you the full read-out.

Table 3.3 The findings of the PCA

Number of variables	Extra information captured	Total data captured
1	28%	28%
2	17%	45%
3	14%	59%
4	10%	69%
5	8%	77%
6	7%	84%
7	5%	89%
8 - 11	…	100%

An elbow shape in the plot suggests that five variables can hold most of the information found inside the data. You could argue for a cut-off at six or seven variables instead, but we're going to opt for a simpler data set versus one with less variance in data against the original data set.

At this point, we could go ahead and see if the original data set recoded with five latent variables is good enough to predict the quality of the wine accurately, but before we do, we'll see how we might identify what they represent.

INTERPRETING THE NEW VARIABLES

With the initial decision made to reduce the data set from 11 original variables to 5 latent variables, we can check to see whether it's possible to interpret or name them based on their relationships with the originals. Actual names are easier to work with than codes such as lv1, lv2, and so on. We can add the line of code in the following listing to generate a table that shows how the two sets of variables correlate.

Listing 3.7 Showing PCA components in a Pandas data frame

```
pd.DataFrame(results.components_, columns=list(
➥   [u'fixed acidity', u'volatile acidity', u'citric acid', u'residual sugar',
➥    u'chlorides', u'free sulfur dioxide',  u'total sulfur dioxide', u'density',
➥    u'pH', u'sulphates',  u'alcohol']))
```

The rows in the resulting table (table 3.4) show the mathematical correlation. Or, in English, the first latent variable lv1, which captures approximately 28% of the total information in the set, has the following formula.

```
Lv1 = (fixed acidity * 0.489314) + (volatile acidity * -0.238584) + … +
(alcohol * -0.113232)
```

Table 3.4 How PCA calculates the 11 original variables' correlation with 5 latent variables

	Fixed acidity	Volatile acidity	Citric acid	Residual sugar	Chlorides	Free sulfur dioxide	Total sulfur dioxide	Density	pH	Sulphates	Alcohol
0	0.489314	-0.238584	0.463632	0.146107	0.212247	-0.036158	0.023575	0.395353	-0.438520	0.242921	-0.113232
1	-0.110503	0.274930	-0.151791	0.272080	0.148052	0.513567	0.569487	0.233575	0.006711	-0.037554	-0.386181
2	0.123302	0.449963	-0.238247	-0.101283	0.092614	-0.428793	-0.322415	0.338871	-0.057697	-0.279786	-0.471673
3	-0.229617	0.078960	-0.079418	-0.372793	0.666195	-0.043538	-0.034577	-0.174500	-0.003788	0.550872	-0.122181
4	0.082614	-0.218735	0.058573	-0.732144	-0.246501	0.159152	0.222465	-0.157077	-0.267530	-0.225962	-0.350681

Giving a useable name to each new variable is a bit trickier and would probably require consultation with an actual wine expert for accuracy. However, as we don't have a wine expert on hand, we'll call them the following (table 3.5).

Table 3.5 Interpretation of the wine quality PCA-created variables

Latent variable	Possible interpretation
0	Persistent acidity
1	Sulfides
2	Volatile acidity
3	Chlorides
4	Lack of residual sugar

We can now recode the original data set with only the five latent variables. Doing this is data preparation again, so we revisit step three of the data science process: data preparation. As mentioned in chapter 2, the data science process is a recursive one and this is especially true between step three: data preparation and step 4: data exploration.

Table 3.6 shows the first three rows with this done.

Table 3.6 The first three rows of the Red Wine Quality Data Set recoded in five latent variables

	Persistent acidity	Sulfides	Volatile acidity	Chlorides	Lack of residual sugar
0	-1.619530	0.450950	**1.774454**	0.043740	-0.067014
1	-0.799170	1.856553	0.911690	0.548066	0.018392
2	**2.357673**	-0.269976	-0.243489	-0.928450	**1.499149**

Already we can see high values for wine 0 in volatile acidity, while wine 2 is particularly high in persistent acidity. Don't sound like good wines at all!

COMPARING THE ACCURACY OF THE ORIGINAL DATA SET WITH LATENT VARIABLES

Now that we've decided our data set should be recoded into 5 latent variables rather than the 11 originals, it's time to see how well the new data set works for predicting the quality of wine when compared to the original. We'll use the Naïve Bayes Classifier algorithm we saw in the previous example for supervised learning to help.

Let's start by seeing how well the original 11 variables could predict the wine quality scores. The following listing presents the code to do this.

Listing 3.8 Wine score prediction before principal component analysis

```
from sklearn.cross_validation import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import confusion_matrix
import pylab as plt

gnb = GaussianNB()
fit = gnb.fit(X,y)
pred = fit.predict(X)
print confusion_matrix(pred,y)
print confusion_matrix(pred,y).trace()
```

Use Gaussian distribution Naïve Bayes classifier for estimation.

Fit data.

Predict data for unseen data.

Study confusion matrix.

Count of all correctly classified cases: all counts on trace or diagonal summed up after analyzing confusion matrix. We can see the Naïve Bayes classifier scores 897 correct predictions out of 1599.

Now we'll run the same prediction test, but starting with only 1 latent variable instead of the original 11. Then we'll add another, see how it did, add another, and so on to see how the predictive performance improves. The following listing shows how this is done.

Listing 3.9 Wine score prediction with increasing number of principal components

Array will be filled with correctly predicted observations

Loops through first 10 detected principal components

Instantiate PCA model with 1 component (first iteration) up to 10 components (in 10th iteration)

Fit PCA model on x-variables (features)

Z is result in matrix form (actually an array filled with arrays)

Use Gaussian distribution Naïve Bayes classifier for estimation

The actual prediction itself using the fitted model

At end of each iteration we append number of correctly classified observations

Printing this array we can see how after each iteration, new count of correctly classified observations is appended

Easier to see when array plotted

Plot shown

```
predicted_correct = []
for i in range(1,10):
    model = PCA(n_components = i)
    results = model.fit(X)
    Z = results.transform(X)
    fit = gnb.fit(Z,y)
    pred = fit.predict(Z)
    predicted_correct.append(confusion_matrix(pred,y).trace())
    print predicted_correct
plt.plot(predicted_correct)
plt.show()
```

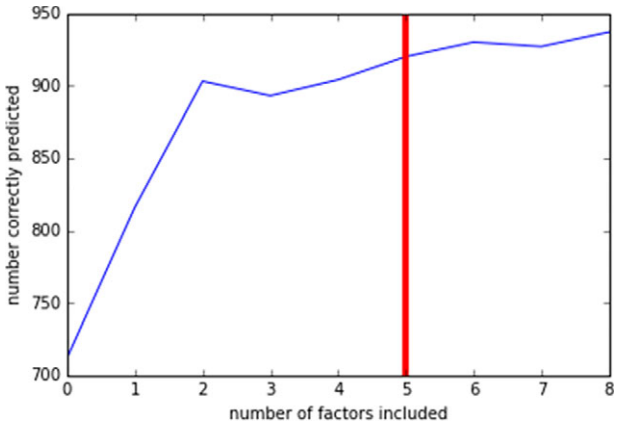


Figure 3.9 The results plot shows that adding more latent variables to a model (x-axis) greatly increases predictive power (y-axis) up to a point but then tails off. The gain in predictive power from adding variables wears off eventually.

The resulting plot is shown in figure 3.9.

The plot in figure 3.9 shows that with only 3 latent variables, the classifier does a better job of predicting wine quality than with the original 11. Also, adding more latent variables beyond 5 doesn't add as much predictive power as the first 5. This shows our choice of cutting off at 5 variables was a good one, as we'd hoped.

We looked at how to group similar variables, but it's also possible to group observations.

GROUPING SIMILAR OBSERVATIONS TO GAIN INSIGHT FROM THE DISTRIBUTION OF YOUR DATA

Suppose for a moment you're building a website that recommends films to users based on preferences they've entered and films they've watched. The chances are high that if they watch many horror movies they're likely to want to know about new horror movies and not so much about new teen romance films. By grouping together users who've watched more or less the same films and set more or less the same preferences, you can gain a good deal of insight into what else they might like to have recommended.

The general technique we're describing here is known as *clustering*. In this process, we attempt to divide our data set into observation subsets, or *clusters*, wherein observations should be similar to those in the same cluster but differ greatly from the observations in other clusters. Figure 3.10 gives you a visual idea of what clustering aims to achieve. The circles in the top left of the figure are clearly close to each other while being farther away from the others. The same is true of the crosses in the top right.

Scikit-learn implements several common algorithms for clustering data in its `sklearn.cluster` module, including the k-means algorithm, affinity propagation, and spectral clustering. Each has a use case or two for which it's more suited,[5] although

[5] You can find a comparison of all the clustering algorithms in Scikit-learn at http://scikit-learn.org/stable/modules/clustering.html.

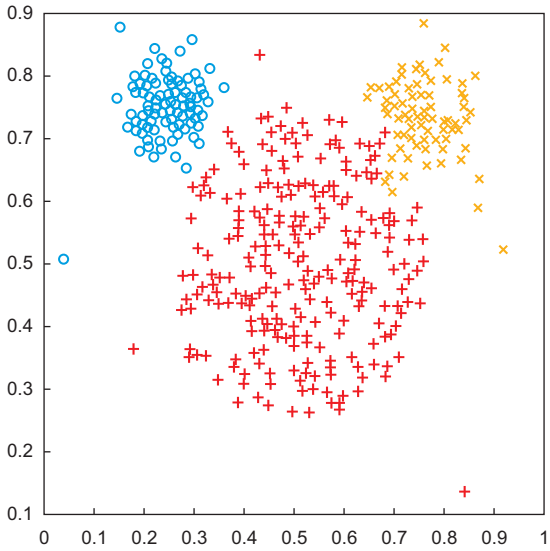


Figure 3.10 The goal of clustering is to divide a data set into "sufficiently distinct" subsets. In this plot for instance, the observations have been divided into three clusters.

k-means is a good general-purpose algorithm with which to get started. However, like all the clustering algorithms, you need to specify the number of desired clusters in advance, which necessarily results in a process of trial and error before reaching a decent conclusion. It also presupposes that all the data required for analysis is available already. What if it wasn't?

Let's look at the actual case of clustering irises (the flower) by their properties (sepal length and width, petal length and width, and so on). In this example we'll use the k-means algorithm. It's a good algorithm to get an impression of the data but it's sensitive to start values, so you can end up with a different cluster every time you run the algorithm unless you manually define the start values by specifying a seed (constant for the start value generator). If you need to detect a hierarchy, you're better off using an algorithm from the class of hierarchical clustering techniques.

One other disadvantage is the need to specify the number of desired clusters in advance. This often results in a process of trial and error before coming to a satisfying conclusion.

Executing the code is fairly simple. It follows the same structure as all the other analyses except you don't have to pass a target variable. It's up to the algorithm to learn interesting patterns. The following listing uses an iris data set to see if the algorithm can group the different types of irises.

Listing 3.10 Iris classification example

Print first 5 observations of data frame to screen; now we can clearly see 4 variables: sepal length, sepal width, petal length, and petal width.

Add another variable called "cluster" to data frame. This indicates the cluster membership of every flower in data set.

Load in iris (flowers) data of Scikit-learn.

Fit model to data. All variables are considered independent variables; unsupervised learning has no target variable (y).

Transform iris data into Pandas data frame.

```
import sklearn
from sklearn import cluster
import pandas as pd

data = sklearn.datasets.load_iris()
X = pd.DataFrame(data.data, columns = list(data.feature_names))
print X[:5]
model   = cluster.KMeans(n_clusters=3, random_state=25)
results = model.fit(X)
X["cluster"] = results.predict(X)
X["target"] = data.target
X["c"] = "lookatmeIamimportant"
print X[:5]
classification_result = X[["cluster",
     "target","c"]].groupby(["cluster","target"]).agg("count")
print(classification_result)
```

Let's finally add a target variable (y) to the data frame.

Adding a variable c is just a little trick we use to do a count later. The value here is arbitrary because we need a column to count the rows.

Three parts to this code. First we select the cluster, target, and c columns. Then we group by the cluster and target columns. Finally, we aggregate the row of the group with a simple count aggregation.

Initialize a k-means cluster model with 3 clusters. The random_state is a random seed; if you don't put it in, the seed will also be random. We opt for 3 clusters because we saw in the last listing this might be a good compromise between complexity and performance.

The matrix this classification result represents gives us an indication of whether our clustering was successful. For cluster 0, we're spot on. On clusters 1 and 2 there has been a slight mix-up, but in total we only get 16 (14+2) misclassifications out of 150.

Figure 3.11 shows the output of the iris classification.

This figure shows that even without using a label you'd find clusters that are similar to the official iris classification with a result of 134 (50+48+36) correct classifications out of 150.

You don't always need to choose between supervised and unsupervised; sometimes combining them is an option.

		c
cluster	target	
0	0	50
1	1	48
	2	14
2	1	2
	2	36

Figure 3.11 Output of the iris classification

3.4 Semi-supervised learning

It shouldn't surprise you to learn that while we'd like all our data to be labeled so we can use the more powerful supervised machine learning techniques, in reality we often start with only minimally labeled data, if it's labeled at all. We can use our unsupervised machine learning techniques to analyze what we have and perhaps add labels to the data set, but it will be prohibitively costly to label it all. Our goal then is to train our predictor models with as little labeled data as possible. This is where semi-supervised learning techniques come in—hybrids of the two approaches we've already seen.

Take for example the plot in figure 3.12. In this case, the data has only two labeled observations; normally this is too few to make valid predictions.

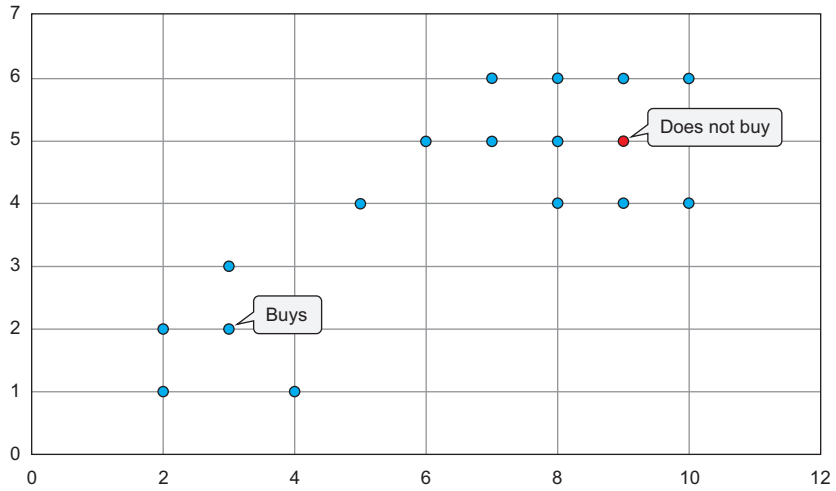


Figure 3.12 This plot has only two labeled observations—too few for supervised observations, but enough to start with an unsupervised or semi-supervised approach.

A common semi-supervised learning technique is *label propagation*. In this technique, you start with a labeled data set and give the same label to similar data points. This is similar to running a clustering algorithm over the data set and labeling each cluster based on the labels they contain. If we were to apply this approach to the data set in figure 3.12, we might end up with something like figure 3.13.

One special approach to semi-supervised learning worth mentioning here is *active learning*. In active learning the program points out the observations it wants to see labeled for its next round of learning based on some criteria you have specified. For example, you might set it to try and label the observations the algorithm is least certain about, or you might use multiple models to make a prediction and select the points where the models disagree the most.

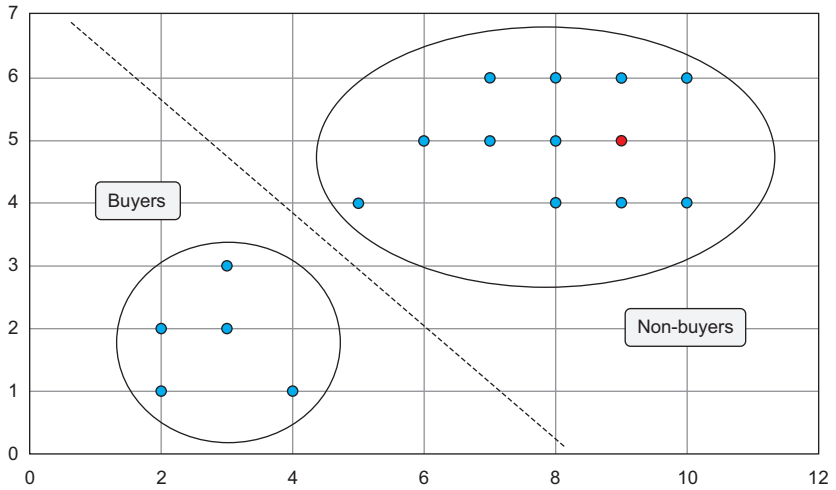


Figure 3.13 The previous figure shows that the data has only two labeled observations, far too few for supervised learning. This figure shows how you can exploit the structure of the underlying data set to learn better classifiers than from the labeled data only. The data is split into two clusters by the clustering technique; we only have two labeled values, but if we're bold we can assume others within that cluster have that same label (buyer or non-buyer), as depicted here. This technique isn't flawless; it's better to get the actual labels if you can.

With the basics of machine learning at your disposal, the next chapter discusses using machine learning within the constraints of a single computer. This tends to be challenging when the data set is too big to load entirely into memory.

3.5 Summary

In this chapter, you learned that

- Data scientists rely heavily on techniques from statistics and machine learning to perform their modeling. A good number of real-life applications exist for machine learning, from classifying bird whistling to predicting volcanic eruptions.
- The modeling process consists of four phases:
  1. *Feature engineering, data preparation, and model parameterization*—We define the input parameters and variables for our model.
  2. *Model training*—The model is fed with data and it learns the patterns hidden in the data.
  3. *Model selection and validation*—A model can perform well or poorly; based on its performance we select the model that makes the most sense.
  4. *Model scoring*—When our model can be trusted, it's unleashed on new data. If we did our job well, it will provide us with extra insights or give us a good prediction of what the future holds.

- The two big types of machine learning techniques
  1. *Supervised*—Learning that requires labeled data.
  2. *Unsupervised*—Learning that doesn't require labeled data but is usually less accurate or reliable than supervised learning.
- Semi-supervised learning is in between those techniques and is used when only a small portion of the data is labeled.
- Two case studies demonstrated supervised and unsupervised learning, respectively:
  1. Our first case study made use of a Naïve Bayes classifier to classify images of numbers as the number they represent. We also took a look at the confusion matrix as a means to determining how well our classification model is doing.
  2. Our case study on unsupervised techniques showed how we could use principal component analysis to reduce the input variables for further model building while maintaining most of the information.