SECOND EDITION

SAMPLE CHAPTER

Algorithms of the Intelligent Web

Douglas G. McIlwraith
Haralambos Marmanis
Dmitry Babenko

FOREWORD BY Yike Guo

MANNING

Algorithms of the Intelligent Web

by Douglas G. McIlwraith
Haralambos Marmanis
Dmitry Babenko

Chapter 2

brief contents

2 Extracting structure from data: clustering and transforming your data

This chapter covers

- Features and the feature space
- Expectation maximization—a way of training algorithms
- Transforming your data axes to better represent your data

In the previous chapter, you got your feet wet with the concept of intelligent algorithms. From this chapter onward, we're going to concentrate on the specifics of machine-learning and predictive-analytics algorithms. If you've ever wondered what types of algorithms are out there and how they work, then these chapters are for you!

This chapter is specifically about the structure of data. That is, given a dataset, are there certain patterns and rules that describe it? For example, if we have a data set of the population and their job titles, ages, and salaries, are there any general rules or patterns that could be used to simplify the data? For instance, do higher

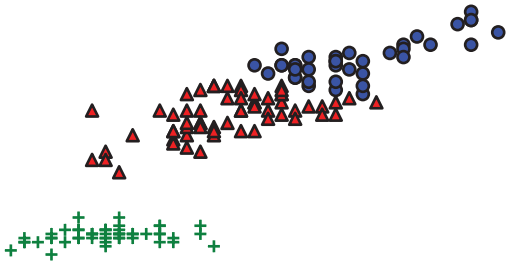

Figure 2.1 Visualizing structure in data. x- and y-axis values are arbitrary and unimportant. This data shows three different classes of data, all colored differently. You see that the x and y values appear to cluster around certain areas of the x, y plot dependent on their class. You also see that higher values of x correlate to higher values of y regardless of the class. Each of these properties relates to the structure of the data, and we'll cover each in turn in the remainder of this chapter.

ages correlate with higher salaries? Is a larger percentage of the wealth present in a smaller percentage of the population? If found, these generalizations can be extracted directly either to provide evidence of a pattern or to represent the dataset in a smaller, more compact data file. These two use cases are the purpose of this chapter, and figure 2.1 provides a visual representation.

> **NOTE TO PRINT BOOK READERS: COLOR GRAPHICS** Many graphics in this book are best viewed in color. The eBook versions display the color graphics, so they should be referred to as you read. To get your free eBook in PDF, ePub, and Kindle formats, go to www.manning.com/books/algorithms-of-the-intelligent-web-second-edition to register your print book.

We'll embark on this subject by laying down some fundamental terms and defining the meaning of structure as it relates to data. We'll also discuss the concepts of bias and noise, which can color your collected data in unexpected ways. You'll also find a discussion about the curse of dimensionality and the feature space. Simply put, this helps us reason about the relationship between the number of data features, the number of data points, and the phenomenon that we're trying to capture in an intelligent algorithm.

We'll move on to talk about a specific method of capturing structure, known as *clustering*, which seeks to group data points together based on their similarity—equivalent to coloring the data as in figure 2.1. Clustering has a wide range of applications, because we often want to know which things are similar to each other. For example, a business might want to find customers who are similar to each other in the hope that the purchase characteristics of one will be similar to those of the others. This might be useful if you're trying to identify your best clients and to grow your user base by locating more people who are like them. Whatever your reason for clustering, there are many ways to perform it, and we'll zero in on a specific type of clustering method known as *k-means*. K-means seeks to partition its input data into k distinct regions or clusters. Although conceptually simple, the k-means algorithm is widely used in practice, and we'll provide you with all the required code to get your very own implementation off the ground using scikit-learn. As part of our discussion of k-means, you'll see how the clusters are learned through the use of an iterative training algorithm called

expectation maximization (EM). This key concept will be revisited again in this chapter when we discuss our second clustering method using the *Gaussian mixture model* (GMM).

Using a GMM to cluster can be seen as an extension of the k-means algorithm. It's possible to achieve exactly the same results with GMM and k-means if you provide certain limitations to your GMM. We'll discuss this in much more detail in the coming sections, but for now, you can think of GMM as a slightly more expressive model that assumes and captures some additional information about the distribution of data points. This model is also trained using the EM algorithm, and we'll investigate and discuss how and why these two versions of EM differ slightly.

In the final section, we'll review an important method for investigating the structure of data and for reducing the total number of features in your dataset without sacrificing the information contained in your data. This topic is equivalent to understanding how your variables vary together in a data set. In figure 2.1, it's equivalent to understanding that x increases with y regardless of the class of data. The method we'll cover is known as *principal component analysis* (PCA). Used on its own, this algorithm uncovers the principal directions of variance in your data—helping you understand which of your features are important and which are not. It can also be used to map your data into a smaller feature space—that is, one with fewer data features—without losing much of the information it captures. Consequently, it can be used as a preprocessing step to your clustering algorithms (or any other intelligent algorithm). This can often impact the training time of your algorithm positively without compromising the algorithm's efficacy. So without further delay, let's investigate the world of data and structure!

2.1 Data, structure, bias, and noise

For the purposes of this book, *data* is any information that can be collected and stored. But not all data is created equal. For intelligent algorithms on the web, data is collected in order to make predictions about a user's online behavior. With this in mind, let's add a little color to the terms *structure*, *bias*, and *noise*.

In general, when we talk about structure, we're referring to some implicit grouping, ordering, pattern, or correlation in the data. For example, if we were to collate the heights of all individuals in the UK, we might find two distinct groups relating to the average heights of male and female individuals. Bias might be introduced to the data if the person or people collecting the data consistently underreported their measurements, whereas noise might be introduced if those same people were careless and imprecise with their measurements.

To investigate these concepts further, we're going to explore some real data known as the *Iris dataset*. This commonly used dataset originated in 1936 and presents 150 data samples from 3 species of flower. For each data item, four measurements (features) are available. Let's begin by interrogating the data; see the following listing.

Listing 2.1 Interrogating the Iris dataset in scikit-learn (interactive shell)

```
>>> import numpy as np
>>> from sklearn import datasets
>>>
>>> iris = datasets.load_iris()
>>> np.array(zip(iris.data,iris.target))[0:10]
array([[array([ 5.1,  3.5,  1.4,  0.2]), 0],
       [array([ 4.9,  3. ,  1.4,  0.2]), 0],
       [array([ 4.7,  3.2,  1.3,  0.2]), 0],
       [array([ 4.6,  3.1,  1.5,  0.2]), 0],
       [array([ 5. ,  3.6,  1.4,  0.2]), 0],
       [array([ 5.4,  3.9,  1.7,  0.4]), 0],
       [array([ 4.6,  3.4,  1.4,  0.3]), 0],
       [array([ 5. ,  3.4,  1.5,  0.2]), 0],
       [array([ 4.4,  2.9,  1.4,  0.2]), 0],
       [array([ 4.9,  3.1,  1.5,  0.1]), 0]], dtype=object)
```

Imports scikit-learn datasets

Loads in the Iris dataset

Creates an array of both the data and the target, and displays the first 10 lines

Contents of the first 10 rows of the array displayed

Listing 2.1 presents an interactive Python session using scikit-learn to interrogate the Iris dataset. After firing up a Python shell, we imported NumPy and the datasets package, before loading in the Iris dataset. We used the `zip` method to create a 2-D array in which `iris.data` appeared alongside `iris.target`, and we selected the first 10 entries in the array.

But what are `iris.data` and `iris.target`? And what did we create? Well, the `iris` object is an instance of `sklearn.datasets.base.Bunch`. This is a dictionary-like object with several attributes (for more details, see the scikit-learn documentation). `Iris.data` is a 2-D array of feature values, and `iris.target` is a 1-D array of classes. For each row in listing 2.1, we printed out an array of the attributes (or *features*) of the flower, along with the class of the flower. But what do these feature values and classes relate to? Each feature relates to the length and width of the petal and sepal, whereas the class relates to the type of flower that was sampled. The next listing provides a summary of this information. Note that the features are presented in the same order as in listing 2.1.

Listing 2.2 Interrogating the Iris dataset in scikit-learn (interactive shell, continued)

```
>>> print(iris.DESCR)
Iris Plants Database

Notes
-----
Data Set Characteristics:
    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
```

```
            - Iris-Setosa
            - Iris-Versicolour
            - Iris-Virginica
    :Summary Statistics:
    ============== ==== ==== ======= ===== ====================
                    Min  Max   Mean    SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length:   4.3  7.9   5.84   0.83    0.7826
    sepal width:    2.0  4.4   3.05   0.43   -0.4194
    petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)
    petal width:    0.1  2.5   1.20   0.76    0.9565  (high!)
    ============== ==== ==== ======= ===== ====================
    :Missing Attribute Values: None
    :Class Distribution: 33.3% for each of 3 classes.
    :Creator: R.A. Fisher
    :Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
    :Date: July, 1988
```

Using the `DESCR` attribute, you can obtain and print to the console some information that has been recorded about the Iris dataset. You can observe that, for each row of `iris.data`, the features consist of the sepal length (cm), the sepal width (cm), the petal length (cm), and the petal width (cm), in that order. The classes of flower, given by `iris.target`, are `Iris-Setosa`, `Iris-Versicolour`, and `Iris-Virginica`. Referring back to listing 2.1, you see that these are encoded by integers. It's easy to understand which class names relate to which integer class labels by using the `target_names` attribute, which stores the values in the same order in which they're encoded. The following code shows that 0 relates to Setosa, 1 to Versicolour, and 2 to Virginica:

```
>>> iris.target_names
array(['setosa', 'versicolour', 'virginica'],
      dtype='|S10')
```

Although we're working with a simple dataset, the concepts that you can learn here are universal among machine-learning algorithms. You'll see later in this chapter how you can draw on machine learning to extract the structure of the data and how different techniques can be adopted to achieve this.

To illustrate the concept of structure, let's try some thought experiments with the Iris dataset. It may be that all Virginicas are much bigger than the other flowers, such that the lengths and widths of the sepals and petals have significantly higher values. It may also be that one of the flowers is characterized by having long, thin petals, whereas the rest have short, rounded petals. At this stage, you don't know what the underlying structure or organization of the data is, but you may wish to employ automatic techniques to uncover this information. You may go further than this. What you might want is for your automated algorithm to uncover this structure and determine these subgroups or clusters and then assign each data point to one of the clusters. This is known as *clustering*.

As mentioned earlier, not all data is equal! Assuming that these measurements characterize the flowers well, and that they somehow capture the uniqueness of the

individual species, there are a few other issues to watch out for. For example, what if the data was collected early in the flower-growing season? What if the flowers were immature and hadn't grown to their full size? This would mean the flowers are systematically different from the overall population—this is known as *bias*. Clusters generated from this data may be fine within the dataset being used, but due to their differences from the rest of the population, the clusters may not generalize well.

Conversely, what if the data-collection task was delegated to many different people? These people would likely perform measurement in different ways and with different levels of care. This would introduce a higher level of randomness to the data, known as *noise*. Depending on the structure of the data, this might be enough to obfuscate any underlying structure or pattern. For maximum generality and performance, both bias and noise should be minimized when collecting data.

2.2 The curse of dimensionality

Although the dataset contains only four features at the moment, you may wonder why we can't collect hundreds or thousands of measurements from each of the flowers and let the algorithm do all the work. Despite the practical challenges in obtaining so many measurements, there are two fundamental problems when collecting data with high dimensionality that are particularly important when finding structure. The first problem is that the large number of dimensions increases the amount of space available for spreading data points. That is, if you keep the number of data points fixed and increase the number of attributes you want to use to describe them, the density of the points in your space decreases exponentially! So, you can wander around in your data space for a long time without being able to identify a formation that's preferable to another one.

The second fundamental problem has a frightening name. It's called the *curse of dimensionality*. In simple terms, it means if you have *any* set of points in high dimensions and you use *any* metric to measure the distance between these points, they'll all come out to be roughly the same distance apart! To illustrate this important effect of dimensionality, let's consider the simple case illustrated in figure 2.2.

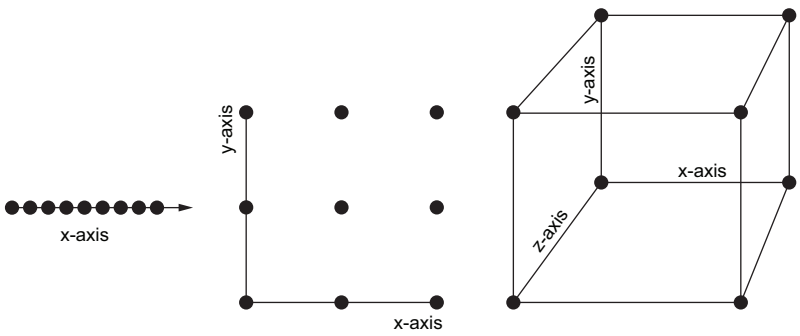

Figure 2.2 The curse of dimensionality: every point tends to have the same distance from any other point.

If you look at figure 2.2 from left to right, you'll see that the dimensionality increases by 1 for each drawing. We start with eight points in one dimension (x-axis) distributed in a uniform fashion: say, between 0 and 1. It follows that the minimum distance we need to traverse from any given point until we meet another point is min(D) = 0.125, whereas the maximum distance is max(D) = 1. Thus, the ratio of min(D) over max(D) is equal to 0.125. In two dimensions, the eight data points are again distributed uniformly, but now we have min(D) = 0.5 and max(D) = 1.414 (along the main diagonal); thus, the ratio of min(D) over max(D) is equal to 0.354. In three dimensions, we have min(D) = 1 and max(D) = 1.732; thus, the ratio of min(D) over max(D) is equal to 0.577. As as the dimensionality continues to increase, the ratio of the minimum distance over the maximum distance approaches the value of 1. This means no matter which direction you look and what distance you measure, it all looks the same!

What that this means practically for you is that the more attributes of the data you collect, the bigger the space of possible points is; it also means the similarity of these points becomes harder to determine. In this chapter, we're looking at the structure of data, and in order to determine patterns and structure in the data, you might think it's useful to collect lots of attributes about the phenomenon under investigation. Indeed it is, but be careful—the number of data points required to generalize patterns grows more quickly than the number of attributes you're using to describe the phenomenon. In practical terms, there's a trade-off to be made: you must collect highly descriptive attributes but not so many that you can't collect enough data to generalize or determine patterns in this space.

2.3 K-means

K-means seeks to partition data into k distinct clusters, characterizing these by their means. Such an outcome is often obtained using Lloyd's algorithm,[1] which is an iterative approach to finding a solution. It is often known as the *k-means algorithm* due to its widespread adoption. Usually, when people say they're applying k-means, they understand that they're using Lloyd's algorithm to solve for k-means. Let's look at a simple example of this algorithm operating on only a single dimension. Consider the number line presented in figure 2.3.

Here we present a dataset containing 13 points. You can clearly see that there's some structure to the data. The leftmost points might be considered to be clustered in the range 0–6, whereas the points to the right might be clustered in the range 9–16. We don't necessarily know that this is a good clustering, but if we were to use our judgment, we might hazard a guess that they're significantly different enough to be considered different groups of points.

This is essentially what all clustering algorithms seek to do: to create groups of data points that we may consider to be clusters of points or structure in the data. We'll dig further into this example before using k-means on some real data, but it's worth

[1] Stuart P. Lloyd, "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory* 28 (1982): 129-137.

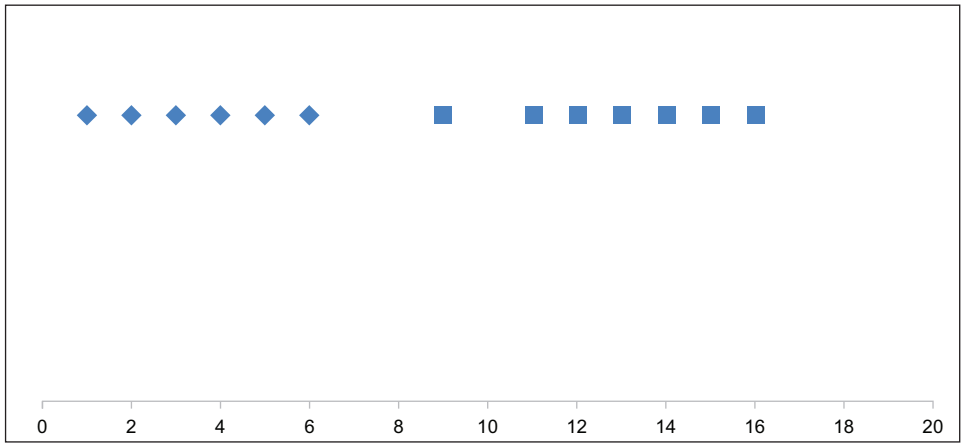

Figure 2.3 A number line containing two possible clusters in a single dimension. An entry indicates the existence of that number in the dataset. Data points represented by the same shape indicate one possible intuitive clustering of the data.

mentioning that this simple one-dimensional example demonstrates the concept without loss of generalization. In reality, nontrivial patterns and structure would be found from higher-dimensional data, subject, of course, to the curse of dimensionality.

Lloyd's algorithm works by iteratively trying to determine the means of the clusters in the dataset. The algorithm takes as a parameter k, the number of clusters that are to be learned; it may also take k estimates of the means of these clusters. These are the initial means of the clusters, but don't worry if these aren't correct yet—the algorithm will iteratively update them in order to provide the best solution it can. Let's revisit the data given in figure 2.3 and see if we can work through an example using this data.

Let's first imagine that we don't know the clusters or the means of these clusters (a sound assumption for a clustering algorithm), but all that we have is the 13 data points. Let's assume that we're also good guessers and we think there are two clusters to the data, so we'll start with k=2. The algorithm then proceeds as follows. First, we must calculate some non-identical estimates for the k means. Let's select these randomly as 1 and 8 for k_1 and k_2, respectively. We denote these by a triangle and a circle in figure 2.4, respectively. These values are our best guesses, and we'll update these as we go.

Now, for each data point in the dataset, we'll assign that point to the nearest cluster. If we work this through, all the points below 4.5 will be assigned to cluster k_1, and all the points above 4.5 will be assigned to cluster k_2. Figure 2.5 shows this with the data points belonging to k_1 shaded green (diamonds, in the print version of this book) and those belonging to k_2 shaded red (squares).

Not a bad start, you might think, but we can do better. If we revisit the data, we see that it looks more like points below 7 belong to the leftmost cluster, k_1, and points above 7 belong to the rightmost cluster, k_2. In order to make a cluster centroid better

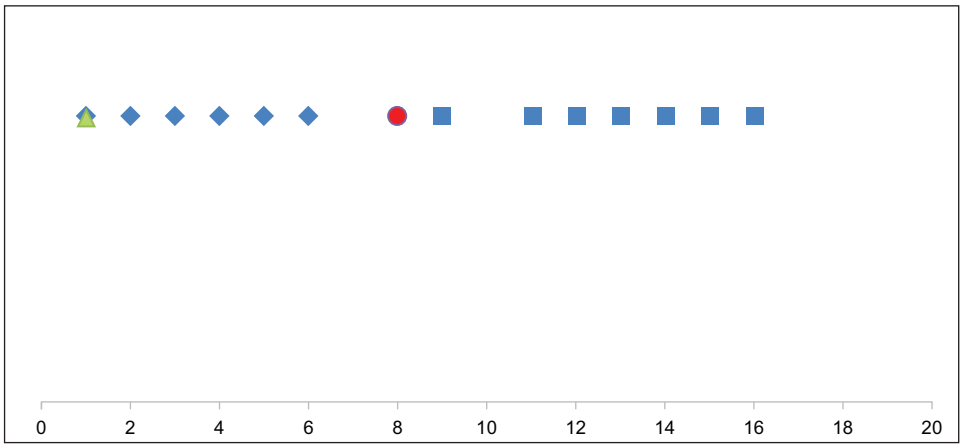

Figure 2.4 Initial assignment of the means of the k clusters. The mean of cluster k_1 is denoted by a triangle, and a circle denotes the mean of cluster k_2.

represent this, new means are calculated based on the average values in that cluster: that is, the new mean for cluster k_1 is given by the mean of the green data points, and the new mean for cluster k_2 is given by the mean of the red data points. Thus, the new means for k_1 and k_2 are 2.5 and 11.2, respectively. You can see the updated means in figure 2.6. Essentially, both cluster means have been dragged up by the data points assigned to them.

We now proceed with the second iteration. As before, for each data point, we assign it to the cluster with the closest centroid. All of our points below approximately 6.8 are now assigned to cluster k_1, and all of our points above this number are

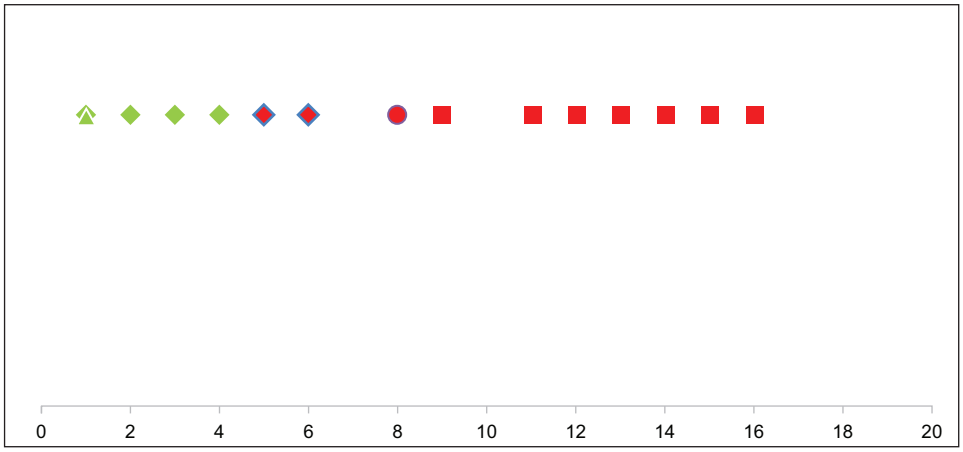

Figure 2.5 Initial assignment of data points to clusters. Those data points assigned to cluster k_1 are shaded green (diamonds), and those assigned to cluster k_2 are shaded red (squares).

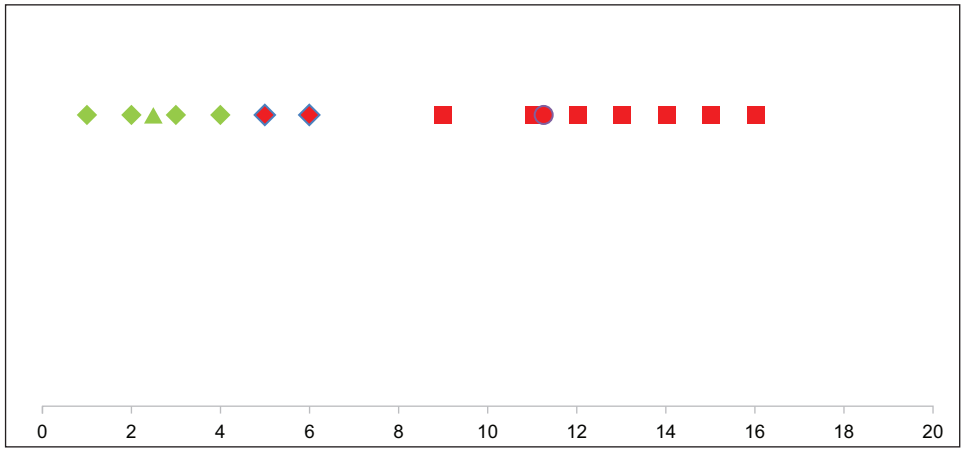

Figure 2.6 Updated means given the new cluster assignment. Cluster centroids have been updated to better represent the data points that have been assigned to them.

assigned to cluster k_2. If we continue as before and calculate the new means of the clusters, this gives centroids of 3.5 and 12.8. At this point, no number of additional iterations will change the assignment of the data points to their clusters, and thus the cluster means won't change. Another way of saying this is that the algorithm has *converged*.[2] Figure 2.7 shows the final assignment.

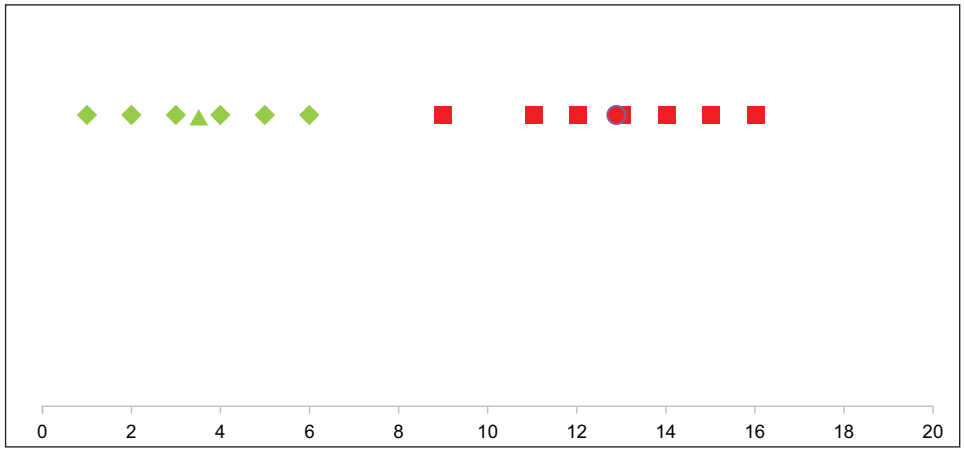

Figure 2.7 Final assignment of the data points to the clusters. Final cluster centroids are given by the triangle and circle, respectively. Data points below 6.8 have been assigned to cluster k_1, and data points above this have been assigned to cluster k_2. Additional iterations of the k-means algorithm won't change this clustering.

[2] The same nomenclature is used when learning the parameters for any algorithm. When a cost function is minimized, practitioners say that the training algorithm has converged. In this book, we'll mostly rely on scikit-learn libraries to reach convergence for us during training.

Note that this final assignment is exactly the means we would have chosen if we had manually performed clustering (that is, we visually determined the clusters and calculated their means); thus the algorithm has automatically determined an intelligent clustering of the data with minimal intervention from the developer!

Be aware that although the algorithm provably converges if you use the Euclidean distance function,[3] there's no guarantee that convergence reaches a global minimum; that is, the final clustering reached may not be the best. The algorithm can be sensitive to initial starting conditions and identify different (local minima) cluster means. Consequently, in application you might consider multiple starting points, rerunning the training several times. One approach then to getting closer to the global minima is to use the average of each resultant centroid group as the starting centroids for a final run. This may put you in a better starting position to reach the best clustering. No such guarantees are given, though.

Listing 2.3 provides the pseudo-code that summarizes these execution steps. In the first case, we must initialize the centroids, one for each k that we're defining at the start of the algorithm. We then enter a loop, which is broken by convergence of the algorithm. In the previous example, we knew intuitively that the algorithm had converged because the cluster centroids and data-point assignment wouldn't change, but in reality, this may not be the case. It's more common to look at the change in the means. If this change drops below a certain tolerance value, you can safely assume that only a small number of points are being reassigned and thus the algorithm has converged.

Listing 2.3 K-means pseudo-code: expectation maximization

```
Initialize centroids
while(centroids not converged):
    For each data item, assign label to that of closest centroid.
    Calculate the centroid using data items assigned to it
```

Expectation step

Maximization step

The expectation step assigns a centroid to every data item based on the Euclidean distance to that centroid (the closest is chosen). The maximization step then recalculates the centroids based on the data items that have been assigned to it; that is, each dimension in the feature space is averaged to obtain the new centroid.

This, in essence, creates a loop in which data is captured by a cluster, which is made to look more like the data it clusters, which then captures more data in the cluster, and so on. Convergence occurs when clusters stop capturing data and become steady. This is a specific implementation of Lloyd's algorithm, and more generally of a class of algorithms known as *expectation maximization*. From here on, we'll refer to this as the k-means algorithm for convenience. We'll revisit the concept of expectation maximization in the following sections of this chapter.

[3] Leon Bottou and Yoshua Bengio, "Convergence Properties of the K-Means Algorithms," *Advances in Neural Information Processing Systems* 7 (1994).

2.3.1 K-means in action

Let's now run the k-means algorithm over a higher-dimensional dataset: the Iris dataset introduced earlier. The following listing shows a simple code snippet to load in the Iris dataset and execute the k-means implementation provided with scikit-learn. It prints to screen the derived clusters of the data and then terminates.

Listing 2.4 The k-means algorithm in action

```
from sklearn.cluster import KMeans
from sklearn import datasets

iris = datasets.load_iris()
X = iris.data
km = KMeans(n_clusters=3)
km.fit(X)

print(km.labels_)
```

Contains only the data, not the labels

Passes in the value of k, which in this case is equal to 3

Given the Iris dataset, the basic example shown here uses k-means to see if, in the absence of any ground truth data relating to the type of the flower, we can detect any intrinsic pattern in the data. In other words, can the algorithm find any structure such that it separates the Iris dataset into three distinct clusters? Try running this now. You'll see that, yes, the algorithm does indeed separate the data into three clusters! This is great, but you'd learn more if you could look at these clusters in comparison to the data. The next listing provides the code to visualize this.

Listing 2.5 Visualizing the output of k-means

```
from sklearn.cluster import KMeans
from sklearn import datasets
from itertools import cycle, combinations
import matplotlib.pyplot as pl

iris = datasets.load_iris()
km = KMeans(n_clusters=3)
km.fit(iris.data)

predictions = km.predict(iris.data)

colors = cycle('rgb')
labels = ["Cluster 1","Cluster 2","Cluster 3"]
targets = range(len(labels))

feature_index=range(len(iris.feature_names))
feature_names=iris.feature_names
combs=combinations(feature_index,2)

f,axarr=pl.subplots(3,2)
axarr_flat=axarr.flat

for comb, axflat in zip(combs,axarr_flat):
        for target, color, label in zip(targets,colors,labels):
                feature_index_x=comb[0]
                feature_index_y=comb[1]
```

```
        axflat.scatter(iris.data[predictions==target,feature_index_x],
        iris.data[predictions==target,feature_index_y],c=color,label=label)
                    axflat.set_xlabel(feature_names[feature_index_x])
                    axflat.set_ylabel(feature_names[feature_index_y])
f.tight_layout()
pl.show()
```

This listing provides the code to plot all 2-D combinations of the Iris feature set, as seen in figure 2.8. Color and shape of the data points denote the cluster to which the point belongs—as discovered by k-means.

In this figure, all combinations of the Iris features are plotted against each other in order to assess the quality of clustering. Note that each point is assigned to the cluster with the centroid closest to it in four-dimensional space. The two-dimensional plots are color-coded such that the color-coding is consistent across the plots.

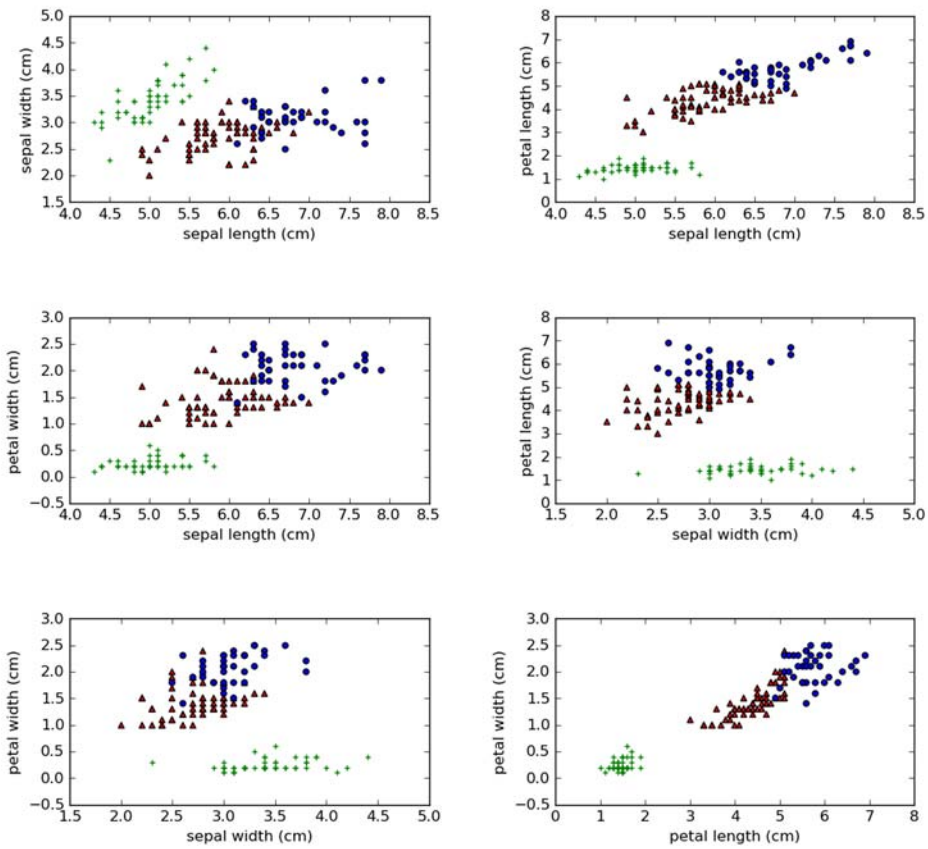

Figure 2.8 K-means clustering of the Iris dataset. This is achieved through the execution of listing 2.5. Note that k-means represents the centroids of the clusters in *n*-dimensional space, where *n* is the number of features (four, in the case of the Iris dataset). For visualization purposes, we plot all combinations of features against each other.

A quick glance over the data seems to suggest that plotting petal length against petal width provides the best way to visualize the data (bottom-right). It also appears that these two variables are highly correlated and that looking at this feature alone can probably separate the three clusters. If this is indeed true, then it begs the question, if a single feature could be used to separate the flowers, why do we need four? Excellent question!

The short answer is that we probably don't, but this wasn't known at the time of collection. With the benefit of hindsight, we might have changed the data that was collected; we did not, however, have that benefit.

In section 2.6, we'll revisit this problem and propose a solution in the form of principal component analysis. This method can generate new axes for your data in which the directions are those with maximal variance in the data. Don't worry about that for now; you can just think of it as distilling all of your features into one or two that have been designed to best illustrate your data.

But before we do this, we're going to continue with a bit more modeling. We'd like to show you another partitional algorithm based on a conceptual modeling approach. In contrast to k-means, this method uses an underlying distribution to model its data—and because the parameters of this distribution must be learned, this type of model is known as a *parametric model*. We'll discuss later how this differs from k-means (a non-parametric model) and under what conditions the two algorithms could be considered equivalent. The algorithm we're going to look at uses the Gaussian distribution as its heart and is known as the *Gaussian mixture model*.

2.4 The Gaussian mixture model

In the previous section, we introduced k-means clustering, which is one of the most widely used approaches for clustering. This non-parametric method for extracting structure has some excellent properties that make it ideal for many situations. In this section, we'll look at a parametric method that uses the Gaussian distribution, known as the Gaussian mixture model (GMM). Both approaches use the expectation maximization (EM) algorithm to learn, so we'll revisit this in more detail in the following sections.

Under certain circumstances, the k-means and GMM approaches can be explained in terms of each other. In k-means, the cluster whose centroid is the closest is assigned directly to a data point, but this assumes that the clusters are scaled similarly and that feature covariance is not dissimilar (that is, you don't have some really long clusters along with really short clusters in the feature space). This is why it often makes sense to normalize your data before using k-means. Gaussian mixtures, however, don't suffer from such a constraint. This is because they seek to model feature covariance for each cluster. If this all seems a little confusing, don't worry! We'll go through these concepts in much more detail in the following sections.

2.4.1 What is the Gaussian distribution?

You may have heard of the Gaussian distribution, sometimes known as the normal distribution, but what exactly is it? We'll give you the mathematical definition shortly, but qualitatively it can be thought of as a distribution that occurs naturally and very frequently.

Using a simple example, if you were to take a random large sample of the population, measure their height, and plot the number of adults in certain height ranges, you'd end up with a histogram that looks something like figure 2.9. This illustrative and wholly fictional data set shows that, from 334 people surveyed, the most common height range lies 2.5 cm on either side of 180 cm.

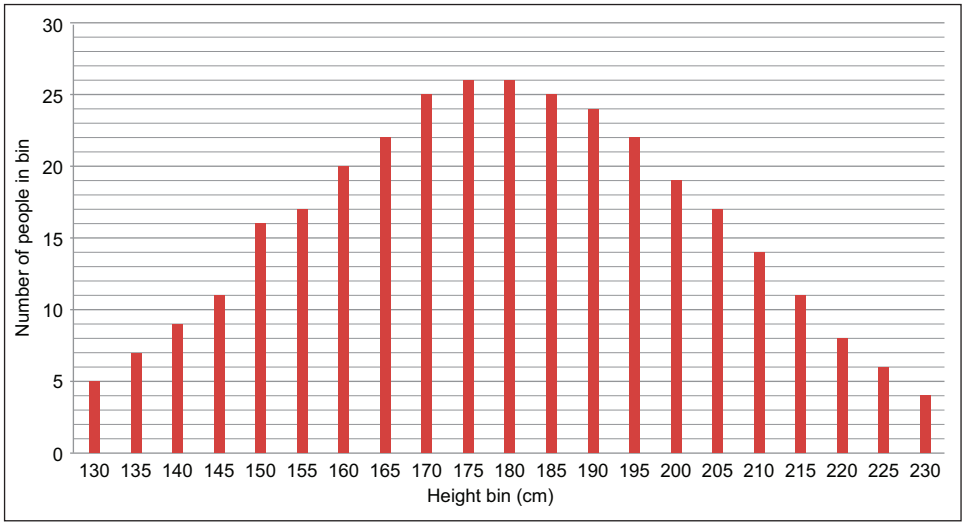

Figure 2.9 Histogram of a normal distribution, in this case the height of 334 fictitious people. The modal (most frequently occurring) bin is centered at 180 cm.

This clearly looks like a Gaussian distribution; but in order to test this assumption, let's use the formulaic definition of a Gaussian to see if we can fit this data through trial and error. The probability density of the normal distribution is given by the following equation:

$$f(x|\mu,\sigma^2) = \frac{1}{\sqrt{2\sigma^2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

This has two parameters, which we need to learn in order to fit this equation to the data: the mean given by the parameter μ and the standard deviation given by the parameter σ. The mean is a measure of the center of the distribution, and if we had to make an informed guess, we might say it was around 180 cm. The standard deviation is a measure of the distribution of the data around the mean. Armed with the knowledge that, for a normal distribution, 95% of the data is always within two standard

deviations from the mean, we might guess that somewhere between 25 and 30 is a good value for this parameter, because most of the collected data does seem to lie between 120 cm and 240 cm.

Recall that the previous equation is a probability density function (PDF). That is, if you plug in a value for x, given a known set of parameters, the function will return a probability density. One more thing, though, before you rush off and implement this! The probability distribution is normalized, such that the area underneath the curve is equal to 1; this is necessary to ensure that returned values are in the range of allowed values for a probability. What this means here is that if we wish to determine the probability of a particular range of values, we must calculate the area under the curve between these two values to determine the likelihood of occurrence. Rather than calculate this area explicitly, we can easily achieve the same results by subtracting the results of the cumulative density function (CDF) for the values of x that define the range. This works because the CDF for a given value of x returns the probability of a sample having a value less than or equal to x.

Let's now return to the original example and see if we can informally evaluate the selected parameters against the collected data. Suppose that we're going to define the data bins in terms of a probability. Each bin corresponds to the probability that, given we select one of the 334 users randomly, the user is in that bin. We can achieve this by dividing the counts on the y-axis by the number of people surveyed (334) to give a probability. This data can be seen in the red plot (the bar on the left in each pair) in figure 2.10. If we now use the cumulative density function to calculate the probabilities of falling in the same bins, with parameters μ=180,σ=28, we can visually inspect how accurate the model is.

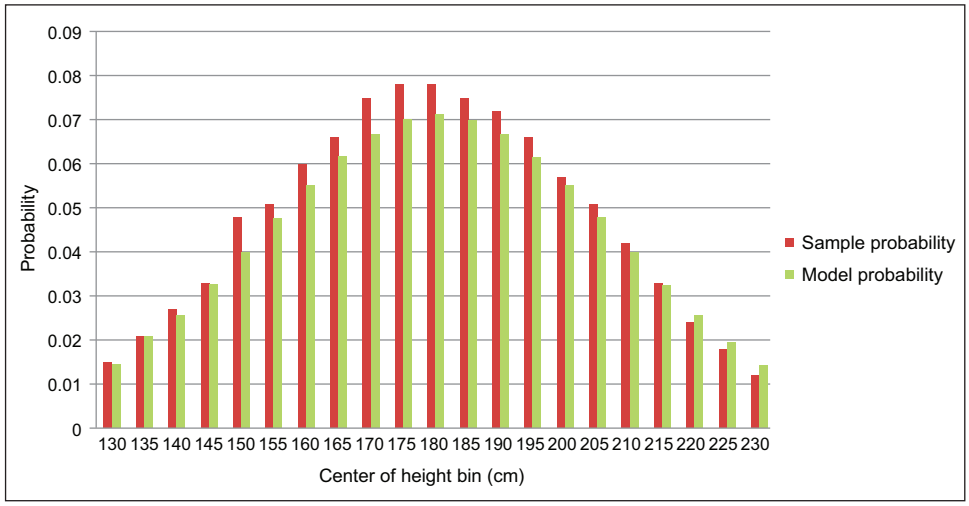

Figure 2.10 Fictional data on the height of 334 people. The sample probability of a given user occurring in a given height range is shown in red (the bar on the left in each pair). The probability from a Gaussian model, with parameters μ=180,σ=28, is shown in green (the bar on the right).

A brief look at figure 2.10 shows that our initial guess of 180 for the mean and 28 for the standard deviation isn't bad. The standard deviation looks reasonable, although it could potentially be decreased slightly. Of course, we could continue to change the parameters here to obtain a better fit, but it might be much better if this was done algorithmically! This process is known as *model training*. We can use the expectation-maximization algorithm to do just this, and you'll see this in the following section.

There's an important distinction to make here about the difference between sample and population distribution. First, we assume that data from the 334 users is representative of the entire population. We're also making an assumption that the underlying distribution that generates the data is Gaussian and that the sample data is drawn from this. As such, it's our best chance at estimating the underlying distribution, but it's just that: an estimate! We're hoping that in the limit (that is, if we collected more and more samples), height tends to a Gaussian, but there will always be an element of uncertainty. The purpose of model training is to minimize this uncertainty under the assumptions of the model.

2.4.2 Expectation maximization and the Gaussian distribution

The previous section presented an intuitive approach to fitting a Gaussian model to a given dataset. In this section, we'll formalize this approach so that model fitting can occur algorithmically. We'll do this by expressing some of the concepts that we used intuitively in a more rigorous manner. When determining whether a model was a good fit, we were looking at the sample probability and seeing whether this was approximated by the model probability. We then changed the model, so that the new model better matched the sample probability. We kept iterating through this process a number of times until the probabilities were very close, after which we stopped and considered the model trained.

We'll do exactly the same thing algorithmically, except we now need something other than a visual evaluation to assess fit. The approach we'll use is to determine the likelihood that the model generates the data; that is, we'll calculate the expectation of the data given the model. We'll then seek to maximize the expectation by updating the model parameters μ,σ. This cycle can then be iterated through until the parameters are changing by only a very small amount at each cycle. Note the similarity to the way in which we trained the k-means algorithm. Whereas in that example we updated only the centroids at the maximization, under a Gaussian model we'll update two parameters: the mean and the variance of the distribution.

2.4.3 The Gaussian mixture model

Now that you understand what a Gaussian distribution is, let's move on to discuss the Gaussian mixture model. This is a simple extension to a Gaussian model that uses multiple Gaussian distributions to model the distribution of data. Let's take a concrete example of this to illustrate. Imagine that instead of measuring the height of users randomly, we now wish to account for both male and female heights in our model.

If we assume that a user can be either male or female, then our Gaussian distribution from earlier is the result of sampling from two separate and overlapping Gaussian distributions. Rather than modeling the space using a single Gaussian, we can now use two (or more!) such distributions:

$$p(x) = \sum_{i=1}^{K} \phi_i \frac{1}{\sqrt{2\sigma_i^2 \pi}} e^{\frac{(x-\mu_i)^2}{2\sigma_i^2}}$$

This equation is extremely similar to the one earlier in the chapter, save for a few details. First, note that instead of a single Gaussian, the probability is made up from the sum of K separate Gaussian distributions, each with its own set of parameters μ,σ. Also note the inclusion of a weighting parameter, φ_i, for each separate distribution. Weights must be positive, and the sum of all weights must equal 1. This ensures that this equation is a true probability density function; in other words, if we integrate this function over its input space, it will be equal to 1.[4]

Returning to the previous example, women may be distributed normally with a lower average height than men, resulting in an underlying generating function drawn from two distributions! See figure 2.11.

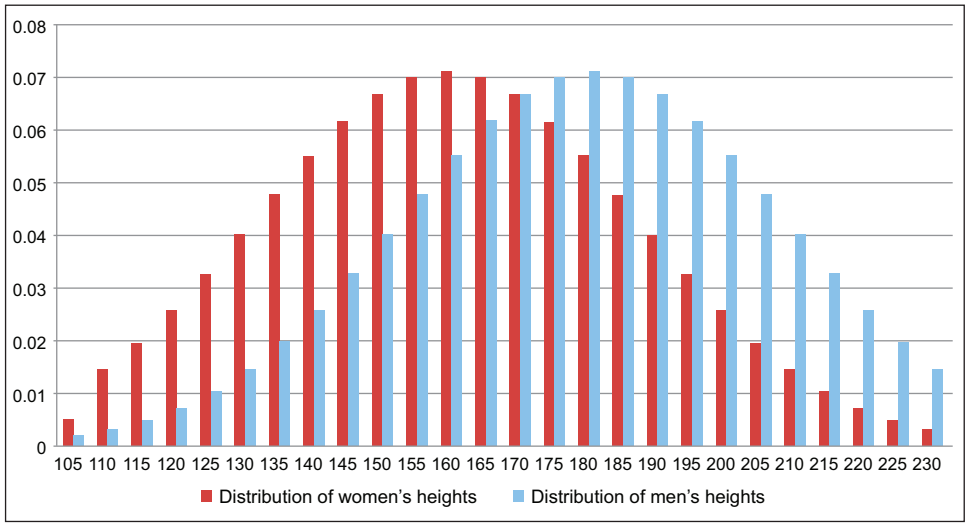

Figure 2.11 Probability distributions of height for men and women. Note that these probabilities are conditioned on the fact that gender is known: so, for example, given that we know a particular person is a woman, her probability of having a height in a particular bucket can be read off the y-axis.

[4] Unlike the previous equation, where the parameters are explicitly represented on the left-hand side of the equation, in this case, the parameters are omitted and implied.

The probabilities given on the y-axis in figure 2.11 are conditioned on the fact that we know the gender of the person. In general, for the purposes of this example, we wouldn't necessarily have that data at hand (maybe it wasn't recorded at measurement time). Thus we'd need to learn not only the parameters of each distribution but also the gender split in the sample population (φ_i). When determining the expectation, this can be calculated by multiplying the probability of being male by the probability of the height bin for a male, and adding this to a similar calculation derived with the data from the distribution over women.

Note that, although doing so is slightly more complicated, we can still use the same technique for model training as specified previously. When working out the expectation (probability that the data is generated by the mixture), we can use the equation presented at the beginning of section 2.4.3. We just need a strategy for updating the parameters to maximize this. Let's use a scikit-learn implementation on the dataset and see how the parameter updates have been calculated.

2.4.4 An example of learning using a Gaussian mixture model

In the rather simple example shown previously, we worked with only single-dimension Gaussians: that is, only a single feature. But Gaussians aren't limited to one dimension! It's fairly easy to extend their mean into a vector and their variance into a covariance matrix, allowing *n*-dimensional Gaussians to cater to an arbitrary number of features. In order to demonstrate this in practice, let's move on from the height example and return to the Iris dataset, which, if you recall, has a total of four features; see the next listing. In the following sections, we'll investigate the implementation of a Gaussian mixture model using scikit-learn and visualize the clusters obtained through its use.

Listing 2.6 Clustering the Iris dataset using a Gaussian mixture model

```
from sklearn.mixture import GMM
from sklearn import datasets
from itertools import cycle, combinations
import matplotlib as mpl
import matplotlib.pyplot as pl
import numpy as np

# make_ellipses method taken from: http://scikit-
# learn.org/stable/auto_examples/mixture/plot_gmm_classifier.html#example-
# mixture-plot-gmm-classifier-py
# Author: Ron Weiss <ronweiss@gmail.com>, Gael Varoquaux
# License: BSD 3 clause

def make_ellipses(gmm, ax, x, y):
    for n, color in enumerate('rgb'):
    row_idx = np.array([x,y])
    col_idx = np.array([x,y])
        v, w = np.linalg.eigh(gmm._get_covars()[n][row_idx[:,None],col_idx])
        u = w[0] / np.linalg.norm(w[0])
        angle = np.arctan2(u[1], u[0])
        angle = 180 * angle / np.pi  # convert to degrees
```

```
        v *= 9
        ell = mpl.patches.Ellipse(gmm.means_[n, [x,y]], v[0], v[1],
                                  180 + angle, color=color)
        ell.set_clip_box(ax.bbox)
        ell.set_alpha(0.5)
        ax.add_artist(ell)

iris = datasets.load_iris()

gmm = GMM(n_components=3,covariance_type='full', n_iter=20)
gmm.fit(iris.data)

predictions = gmm.predict(iris.data)

colors = cycle('rgb')
labels = ["Cluster 1","Cluster 2","Cluster 3"]
targets = range(len(labels))

feature_index=range(len(iris.feature_names))
feature_names=iris.feature_names
combs=combinations(feature_index,2)

f,axarr=pl.subplots(3,2)
axarr_flat=axarr.flat

for comb, axflat in zip(combs,axarr_flat):
    for target, color, label in zip(targets,colors,labels):
        feature_index_x=comb[0]
        feature_index_y=comb[1]
        axflat.scatter(iris.data[predictions==target,feature_index_x],

    iris.data[predictions==target,feature_index_y],c=color,label=label)
        axflat.set_xlabel(feature_names[feature_index_x])
        axflat.set_ylabel(feature_names[feature_index_y])
        make_ellipses(gmm,axflat,feature_index_x,feature_index_y)

pl.tight_layout()
pl.show()
```

The vast majority of the code in listing 2.6 is identical to that of listing 2.5, where we visualized the output from the k-means algorithm. The notable differences are these:

- Using `sklearn.mixture.GMM` instead of `sklearn.cluster.KMeans`
- Defining and using the `make_ellipses` method

You'll notice that, in the former, method initialization is almost exactly the same, aside from some additional parameters. Let's take a look at these. In order to initialize GMM, we pass the following parameters:

- `n_components`—The number of Gaussians in our mixture. The previous example had two.
- `covariance_type`—Restricts the covariance matrix properties and therefore the shape of the Gaussians. Refer to the following documentation for a graphical interpretation of the differences: http://scikit-learn.org/stable/modules/mixture.html.
- `n_iter`—Number of EM iterations to perform.

So in actual fact, the two code snippets are remarkably similar. The only additional decisions that need to be made relate to the relationships of the variances (covariances) that will affect the shape of the resulting clusters.

The remainder of the code presented consists of the `make_ellipses` method. This is a helper method that allows us to present the output of the GMM in a graphical manner. This is taken from the scikit-learn documentation, so you may recognize the code from there. Take a look at the output of listing 2.6 in figure 2.12 before delving into the explanation.

The figure shows the output of our Gaussian mixture model with four dimensions. Three 4-D clusters have been projected onto each set of two dimensions with a representation of the resultant cluster in 2-D through the `make_ellipses` method.

The `make_ellipses` method[5] is conceptually simple. It takes as its arguments the `gmm` object (the trained model), a plot axis (relating to one of the six tiles in figure 2.12), and two indices, `x` and `y`, which relate to the parameters against which the four-

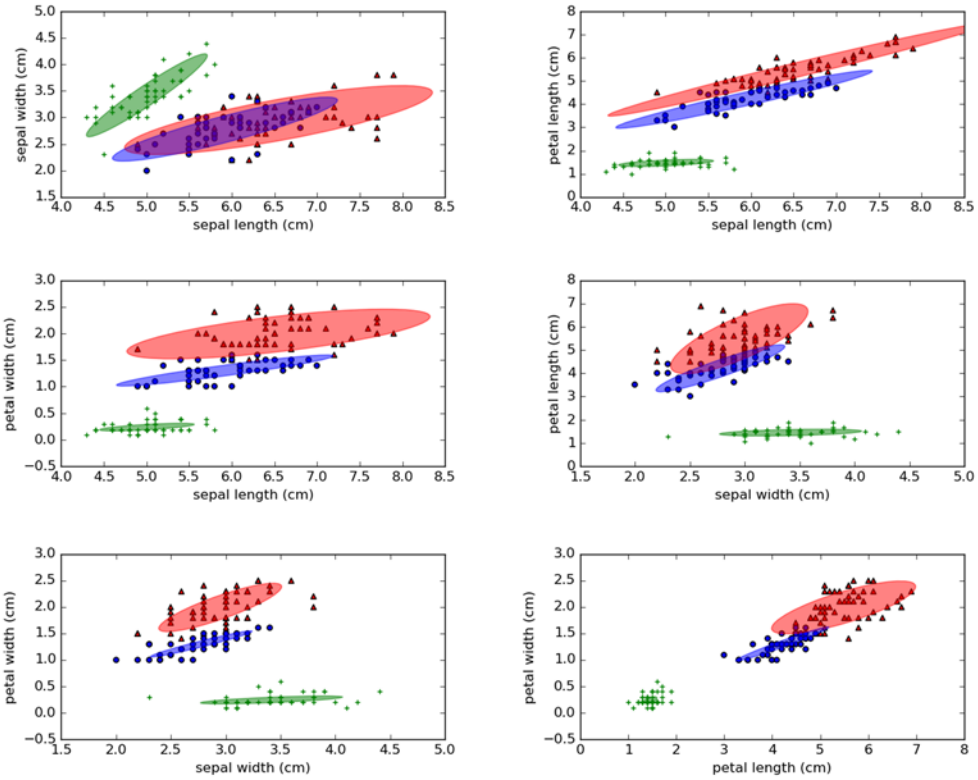

Figure 2.12 Output from listing 2.6. Each panel shows the 4-D Gaussian clustering of the Iris dataset mapped onto two of the axes. The colored clusterings are representations of the underlying four-dimensional Gaussian in two dimensions only.

[5] Ron Weiss and Gael Varoquaux, "GMM classification," *scikit-learn*, http://mng.bz/uKPu.

dimensional plot will be reduced. It returns nothing, but it operates over the axis object, drawing the ellipses in the plot.

> **A word about make_ellipses**
>
> The method `make_ellipses` is derived from `plot_gmm_classifier`, written by Ron Weiss and Gael Varoquaz for scikit-learn. By extracting the covariance matrix in the two dimensions you're plotting, you can find the first and second directions of maximal variance, along with their respective magnitudes. You then use these to orient and draw ellipses relating to the shape of the learned Gaussian. These directions and magnitudes are known as *eigenvectors* and *eigenvalues*, respectively, and will be covered in more detail in section 2.6.

2.5 The relationship between k-means and GMM

K-means can be expressed as a special case of the Gaussian mixture model. In general, the Gaussian mixture is more expressive because membership of a data item to a cluster is dependent on the shape of that cluster, not just its proximity.

The shape of an n-dimensional Gaussian is dependent on the covariance across the dimensions in each cluster. It's possible to obtain the same results with GMM and k-means if additional constraints are placed on the learned covariance matrices.

In particular, if covariance matrices for each cluster are tied together (that is, they all must be the same), and covariances across the diagonal are restricted to being equal, with all other entries set to zero, then you'll obtain spherical clusters of the same size and shape for each cluster. Under such circumstances, each point will always belong to the cluster with the closest mean. Try it and see!

As with k-means, training a Gaussian mixture model with EM can be sensitive to initial starting conditions. If you compare and contrast GMM to k-means, you'll find a few more initial starting conditions in the former than in the latter. In particular, not only must the initial centroids be specified, but the initial covariance matrices and mixture weights must be specified also. Among other strategies,[6] one approach is to run k-means and use the resultant centroids to determine the initial starting conditions for the Gaussian mixture model.

As you can see, there's not a great deal of magic happening here! Both algorithms work similarly, with the main difference being the complexity of the model. In general, all clustering algorithms follow a similar pattern: given a series of data, you can train a model such that the model is in some way a generalization of the data (and hopefully the underlying process generating the data). This training is typically achieved in an iterative fashion, as you saw in the previous examples, with termination occurring when you can't improve the parameters to make the model fit the data any better.

---

[6] Johannes Blomer and Kathrin Bujna, "Simple Methods for Initializing the EM Algorithm for Gaussian Mixture Models," 2013, http://arxiv.org/pdf/1312.5946.pdf.

2.6 Transforming the data axis

So far in this chapter, we've concentrated on clustering data as it's presented in its original feature space. But what if you were to change this feature space into a more appropriate one—perhaps one with fewer features that's more descriptive of the underlying structure?

This is possible using a method known as *principal component analysis* (PCA). This transforms your data axis in such a way that you use the directions of maximal variance in your data as a new data basis (instead of y=0, x=0 that form the standard x-axis and y-axis, respectively). These directions are given by the eigenvectors of your data, and the amount of variance in these directions is given by an associated eigenvalue. In the following sections, we'll discuss these terms in more detail before providing an example of PCA applied to the Iris dataset. You'll see how to reduce the number of features from four to two without losing the ability to cluster the Iris data effectively in the new feature space.

2.6.1 Eigenvectors and eigenvalues

Eigenvectors and eigenvalues[7] are characteristics of a square matrix (a matrix having the same number of rows and columns); that is, they're descriptors of a matrix. What they describe, however, is quite special and quite subtle! In this section, we'll provide some of the intuition behind these terms, before demonstrating the power of these constructs with a dataset. Mathematically, eigenvectors and eigenvalues are defined in the following manner

$$\mathbf{Cv} = \lambda\mathbf{v}$$

where **C** is a square matrix (it has the same number of rows and columns), **v** is an eigenvector, and λ is the corresponding eigenvalue relating to that eigenvector. This may seem a little confusing at first, but with a bit of illustration you should be able to see why this is so powerful.

Let's imagine that the matrix **C** corresponds to some shear transformation in two-dimensional space and is thus a 2 × 2 matrix. Just to hit home with this, given a two-dimensional data set, **C** could be applied to every data point to create a new transformed (sheared) data set.

What you have then is an equation to solve! Can you find a number of vectors (directions) such that after the application of the shear matrix **C**, they remain unchanged save for a difference in magnitude given by λ? The answer is yes, you can! But what does this mean? And why is it important?

Thinking carefully about this, you can see that we're providing a shorthand, or compact form, of the shear transformation. This set of eigenvectors describes the directions unaffected by the application of the shear. But what does the eigenvalue

[7] K. F. Riley, M. P. Hobson, and S. J. Bence, *Mathematical Methods for Physics and Engineering* (Cambridge University Press, 2006).

represent? It represents how much the shear operates in a given direction: that is, the magnitude of that operation. As such, the eigenvalues are a measure of importance of that direction. Thus, you could describe most of the operation of the shear matrix by recording only the eigenvectors with the largest eigenvalues.

2.6.2 Principal component analysis

Principal component analysis[8] uses the eigenvector/eigenvalue decomposition in a specific way in order to obtain the *principal components* of a dataset. Given a dataset with n features, you can work out the covariance matrix, which has the form shown at right.

$$\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ x_{n1} & \cdots & \cdots & x_{nn} \end{bmatrix}$$

The covariance matrix describes the pairwise variance of each feature in the dataset: that is, $x_i x_j$ is the covariance of feature i and feature j. This could be described as a measure of the shape and size of the variation of the dataset. What do you notice about the shape of the matrix? Well, for starters, because you calculate the variance of each feature against all other features, the matrix is always square. That is, the number of rows is the same as the number of columns. In this case, both are equal to n, the number of features in the dataset.

Do you notice anything else about the matrix? That's right: it's symmetrical! Because $x_i x_j = x_j x_i$, you could transpose the matrix (flip it across the diagonal), and the matrix would be the same. Keep these two properties in mind, because they'll come in handy in a few moments.

Let's go back to what the covariance matrix represents. Remember, this is the measure of the shape and size of variation in the dataset. What do you get if you calculate the eigenvectors and eigenvalues of the covariance matrix? One way to understand the covariance matrix is as a relation (via Cholesky decomposition[9]) to a transformation that takes randomly sampled Gaussian data (say, with a covariance matrix where $x_{ij}=1, i=j$ and zero elsewhere) and outputs data distributed in the same manner as the original dataset. Thus, the eigenvectors of the covariance matrix represent the vectors unchanged by this transform variation, otherwise known as the *principal components* of the dataset. But wait, there's more! Because the covariance matrix is symmetrical, the (non-equal) eigenvectors of this matrix are orthogonal! To illustrate, take a look at figure 2.13.

What's great about this decomposition is that you can now express each data point as the linear combination of eigenvectors (due to their orthogonality). For high-dimensional data, this can provide a way to compress your data, because for every data point, you need only store the distance along the eigenvectors of interest. Often, the most variance will be expressed by the first two or three eigenvectors; thus an arbitrarily high-dimensional data point can now be expressed as only a vector with only two or three dimensions.

---

[8] Jonathon Shlens, "A Tutorial on Principal Component Analysis," eprint arXiv:1404.1100, 2014.

[9] Claude Brezinski, "The Life and Work of Andre Cholesky," *Numer. Algorithms* 43 (2006):279-288.

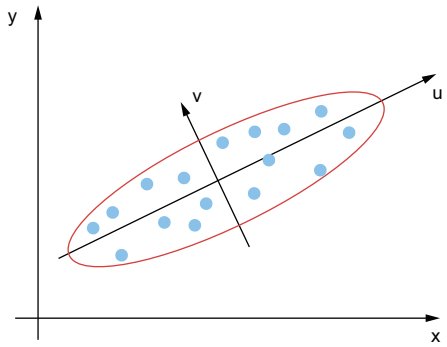

Figure 2.13 The two principal components of a two-dimensional feature set. The eigenvector u has the largest eigenvalue because it provides the largest direction of variance. The vector v has a smaller value in relation. The two vectors are orthogonal: that is, at 90 degrees from each other. In general, for higher dimensions, the dot product of any two pairs of eigenvectors will equal zero.

2.6.3 An example of principal component analysis

Now that you understand eigenvectors, eigenvalues, and PCA, you can get your hands dirty and apply these concepts to some data. As before, we'll use the Iris dataset to demonstrate this concept in scikit-learn. Consider the following listing.

Listing 2.7 Principal component analysis of the Iris dataset

```
import numpy as np
import matplotlib.pyplot as pl

from sklearn import decomposition
from sklearn import datasets
from itertools import cycle

iris = datasets.load_iris()
X = iris.data
Y = iris.target

targets = range(len(iris.target_names))
colors = cycle('rgb')

pca = decomposition.PCA(n_components=2)
pca.fit(X)

X = pca.transform(X)

for target,color in zip(targets,colors):
    pl.scatter(X[Y==target,0],
               X[Y==target,1],
               label=iris.target_names[target],c=color)

pl.legend()
pl.show()
```

1. Initializes a PCA decomposition with two components
2. Fits the data: solves the eigenvector decomposition
3. Transforms the data into the basis. Because we chose to keep only two components, this has two dimensions.
4. For each iris category, plots (in a different color) the coordinates in the new coordinate axis

This PCA transformation of the Iris data yields the output given by figure 2.14. You load in the Iris dataset as before and then initialize a PCA decomposition with two

components. At this point, no decomposition has been performed (for a start, no data has been passed in); you merely prepare the Python objects for the next stage. The next stage is the `fit` method ❷, where the data axis is transformed.

In this single line, the covariance matrix is calculated for the dataset, which in the case of Iris will result in a 4 × 4 square matrix, and the matrix is solved for the eigenvectors. This line performs the heavy lifting, finding the directions of the dataset that account for the maximal variance in the data. The next line obtains the data transformed into the new data axis ❸. The original data is now expressed as a combination of the two eigenvectors with the largest eigenvalues (because you restrict the number of components to two in ❶), which you can now plot as per figure 2.14 ❹. It should now be fairly clear that the x-axis (y=0) and the y-axis (x=0) are the directions (in four dimensions in Iris) of the highest variance and second-highest variance, respectively. The values plotted correspond to the distance along that line of the original data. Thus, you've transformed a 4-D dataset down to a 2-D dataset, while keeping the majority of variance of the data in the new data space.

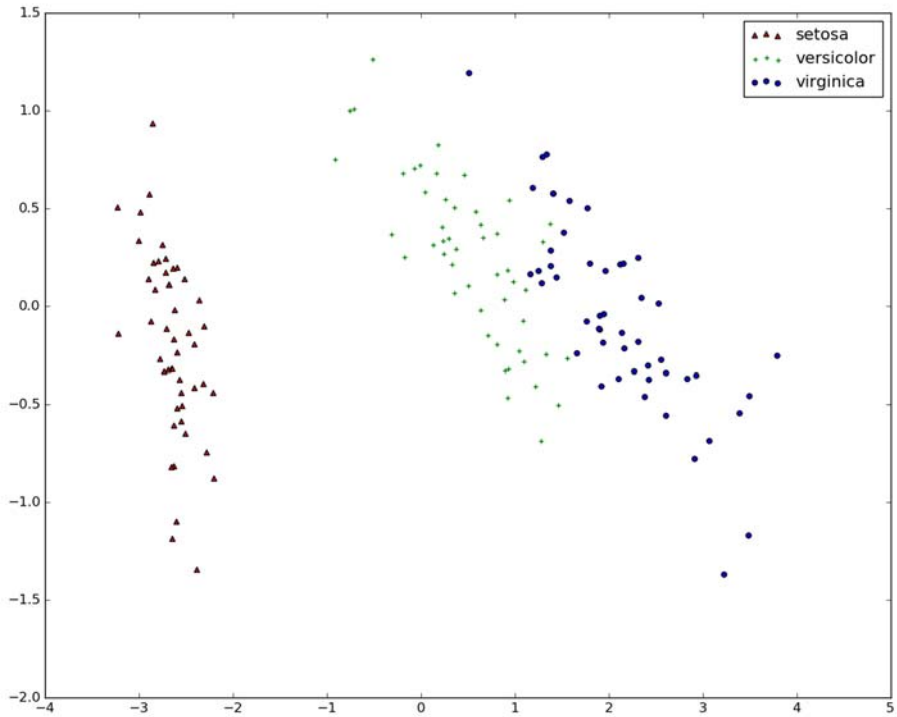

Figure 2.14 Eigenvector decomposition of the Iris dataset using the first two eigenvectors. The x-axis shows the direction in which the data has maximal variance. This is the first eigenvector, and its corresponding eigenvalue is maximal relative to all solutions given this data. The y-axis represents the second eigenvector, and its corresponding eigenvalue has the second-largest value of all solutions given this data. Points in the scatter plot are transformed from the original data into this new axis with respect to these first two eigenvectors.

Principal component analysis is a powerful tool that can help you to calculate a more efficient representation of the data while minimizing loss to that data's discriminatory power. Although we used this on a relatively simple dataset here, it can be, and often is, applied to real-world data as a first pass—as a preprocessing step to the classification and regression problems that we'll spend more time on later in the book. Note, however, that complexity grows faster as the number of features in your dataset grows, so removing redundant and irrelevant features as a step before PCA may pay dividends.

2.7 Summary

- Bias is a systematic offset, applied to all elements of your dataset, whereas noise is a random addition distributed around the true values being measured.
- The feature space is the space containing all feature vectors, for a given number of features. You learned about the curse of dimensionality: the more features collected about a phenomenon, the more data points are required to generalize that phenomenon. The practitioner of machine learning must walk this trade-off carefully. Too many features, and it will be impossible to collect enough data points to generalize the phenomenon under investigation; too few, and the phenomenon is not appropriately captured.
- We looked closely at data structure, covering in detail the k-means algorithm and the Gaussian mixture model. Both of these algorithms perform well in practice, and we explored the relationship between the two. In machine learning, many algorithms can be considered equivalent to others under certain conditions, as is the case here.
- Expectation Maximization (EM) is a general class of algorithms that can be used to fit data to a model through a series of iterations. First, the expectation that the model generated the data is calculated, and then the model is changed by a small amount to make it more likely that the new model is responsible for the data. This has the effect of learning (and approximating) the original distribution of the data that underlies the training samples.
- An algorithm is only as good as its data! If the features collected about a phenomenon aren't sufficiently descriptive, then no amount of machine learning will be able to achieve satisfactory results. The onus is on the practitioner to remove irrelevant and redundant features and to minimize the size of the dataset without sacrificing discriminatory power. This often leads to improved results for the same dataset. Toward this end, we examined principal component analysis, which can be used to distill many features into few without sacrificing the original variance in the dataset.

WEB DEVELOPMENT/PYTHON

Algorithms of the Intelligent Web Second Edition

McIlwraith • Marmanis • Babenko

Free eBook
SEE INSERT

Valuable insights are buried in the tracks web users leave as they navigate pages and applications. You can uncover them by using intelligent algorithms like the ones that have earned Facebook, Google, and Twitter a place among the giants of web data pattern extraction.

Algorithms of the Intelligent Web, Second Edition teaches you how to create machine learning applications that crunch and wrangle data collected from users, web applications, and website logs. In this totally revised edition, you'll look at intelligent algorithms that extract real value from data. Key machine learning concepts are explained with code examples in Python's scikit-learn. This book guides you through algorithms to capture, store, and structure data streams coming from the web. You'll explore recommendation engines and dive into classification via statistical algorithms, neural networks, and deep learning.

What's Inside

- Introduction to machine learning
- Extracting structure from data
- Deep learning and neural networks
- How recommendation engines work

Knowledge of Python is assumed.

Dr. Douglas McIlwraith is a machine learning expert and data science practitioner in the field of online advertising. **Dr. Haralambos Marmanis** is a pioneer in the adoption of machine learning techniques for industrial solutions. **Dmitry Babenko** designs applications for banking, insurance, and supply-chain management.

"Concise descriptions of algorithms with their mathematical foundation and sample code in Python."
—From the Foreword by Yike Guo, Data Science Institute Imperial College London

"This second edition brings fresh life to an all-time classic."
—Marius Butuc, Shopify

"Covers the most essential areas of machine learning application in the real world. A great hands-on approach."
—Radha Ranjan Madhav, Amazon

"Great balance between theory and practice."
—Dike E. Kalu, Fara Frica

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
www.manning.com/books/algorithms-of-the-intelligent-web-second-edition

ISBN-13: 978-1-61729-258-3
ISBN-10: 1-61729-258-3
54499
9 781617 292583

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]