8 Unsupervised methods

This chapter covers

- Using R's clustering functions to explore data and look for similarities
- Choosing the right number of clusters
- Evaluating a clustering
- Using R's association rules functions to find patterns of co-occurrence in data
- Evaluating a set of association rules

The methods that we've discussed in previous chapters build models to predict outcomes. In this chapter, we'll look at methods to discover unknown relationships in data. These methods are called *unsupervised methods*. With unsupervised methods, there's no outcome that you're trying to predict; instead, you want to discover patterns in the data that perhaps you hadn't previously suspected. For example, you may want to find groups of customers with similar purchase patterns, or correlations between population movement and socioeconomic factors. Unsupervised analyses are often not ends in themselves; rather, they're ways of finding relationships and patterns that can be used to build predictive models. In fact, we encourage you to think of unsupervised methods as exploratory—procedures that help

you get your hands in the data—rather than as black-box approaches that mysteriously and automatically give you "the right answer."

In this chapter, we'll look at two classes of unsupervised methods. *Cluster analysis* finds groups in your data with similar characteristics. *Association rule mining* finds elements or properties in the data that tend to occur together.

8.1 Cluster analysis

In cluster analysis, the goal is to group the observations in your data into *clusters* such that every datum in a cluster is more similar to other datums in the same cluster than it is to datums in other clusters. For example, a company that offers guided tours might want to cluster its clients by behavior and tastes: which countries they like to visit; whether they prefer adventure tours, luxury tours, or educational tours; what kinds of activities they participate in; and what sorts of sites they like to visit. Such information can help the company design attractive travel packages and target the appropriate segments of their client base with them.

Cluster analysis is a topic worthy of a book in itself; in this chapter, we'll discuss two approaches. *Hierarchical clustering* finds nested groups of clusters. An example of hierarchical clustering might be the standard plant taxonomy, which classifies plants by family, then genus, then species, and so on. The second approach we'll cover is *k-means*, which is a quick and popular way of finding clusters in quantitative data.

> **Clustering and density estimation**
>
> Historically, cluster analysis is related to the problem of *density estimation*: if you think of your data as living in a large dimensional space, then you want to find the regions of the space where the data is densest. If those regions are distinct, or nearly so, then you have clusters.

8.1.1 Distances

In order to cluster, you need the notions of *similarity* and *dissimilarity*. Dissimilarity can be thought of as distance, so that the points in a cluster are closer to each other than they are to the points in other clusters. This is shown in figure 8.1.

Different application areas will have different notions of distance and dissimilarity. In this section, we'll cover a few of the most common ones:

- Euclidean distance
- Hamming distance
- Manhattan (city block) distance
- Cosine similarity

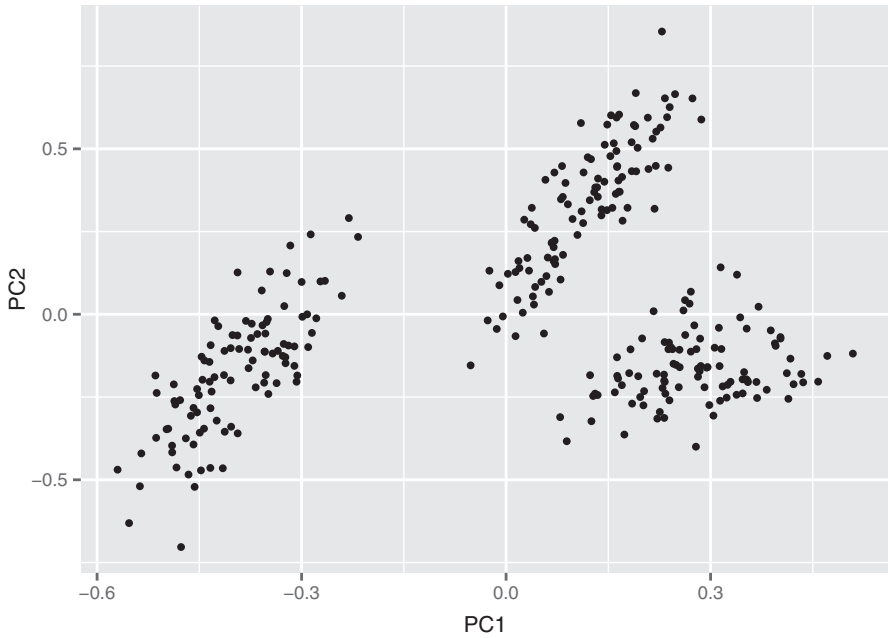

Figure 8.1 **An example of data in three clusters**

EUCLIDEAN DISTANCE

The most common distance is *Euclidean distance*. The Euclidean distance between two vectors `x` and `y` is defined as

```
edist(x, y) <- sqrt((x[1]-y[1])^2 + (x[2]-y[2])^2 + ...)
```

This is the measure people tend to think of when they think of "distance." Optimizing squared Euclidean distance is the basis of k-means. Of course, Euclidean distance only makes sense when all the data is real-valued (quantitative). If the data is categorical (in particular, binary), then other distances can be used.

HAMMING DISTANCE

For categorical variables (`male/female`, or `small/medium/large`), you can define the distance as `0` if two points are in the same category, and `1` otherwise. If all the variables are categorical, then you can use *Hamming distance*, which counts the number of mismatches:

```
hdist(x, y) <- sum((x[1] != y[1]) + (x[2] != y[2]) + ...)
```

Here, `a != b` is defined to have a value of `1` if the expression is true, and a value of `0` if the expression is false.

You can also expand categorical variables to indicator variables (as we discussed in section 7.1.4), one for each level of the variable.

If the categories are ordered (like `small/medium/large`) so that some categories are "closer" to each other than others, then you can convert them to a numerical sequence. For example, (`small/medium/large`) might map to (`1/2/3`). Then you can use Euclidean distance, or other distances for quantitative data.

MANHATTAN (CITY BLOCK) DISTANCE

Manhattan distance measures distance in the number of horizontal and vertical units it takes to get from one (real-valued) point to the other (no diagonal moves):

```
mdist(x, y) <- sum(abs(x[1]-y[1]) + abs(x[2]-y[2]) + ...)
```

This is also known as *L1 distance* (and squared Euclidean distance is *L2 distance*).

COSINE SIMILARITY

Cosine similarity is a common similarity metric in text analysis. It measures the smallest angle between two vectors (the angle `theta` between two vectors is assumed to be between 0 and 90 degrees). Two perpendicular vectors (`theta` = 90 degrees) are the most dissimilar; the cosine of 90 degrees is 0. Two parallel vectors are the most similar (identical, if you assume they're both based at the origin); the cosine of 0 degrees is 1. From elementary geometry, you can derive that the cosine of the angle between two vectors is given by the normalized dot product between the two vectors:

```
dot(x, y) <- sum( x[1]*y[1] + x[2]*y[2] + ... )
cossim(x, y) <- dot(x, y)/(sqrt(dot(x,x)*dot(y,y)))
```

You can turn the cosine similarity into a pseudo distance by subtracting it from 1.0 (though to get an actual metric, you should use `1 - 2*acos(cossim(x,y))/pi`).

Different distance metrics will give you different clusters, as will different clustering algorithms. The application domain may give you a hint as to the most appropriate distance, or you can try several distance metrics. In this chapter, we'll use (squared) Euclidean distance, as it's the most natural distance for quantitative data.

8.1.2 Preparing the data

To demonstrate clustering, we'll use a small dataset from 1973 on protein consumption from nine different food groups in 25 countries in Europe.[1] The goal is to group the countries based on patterns in their protein consumption. The dataset is loaded into R as a data frame called `protein`, as shown in the next listing.

[1] The original dataset can be found at http://mng.bz/y2Vw. A tab-separated text file with the data can be found at https://github.com/WinVector/zmPDSwR/tree/master/Protein/. The data file is called protein.txt; additional information can be found in the file protein_README.txt.

Listing 8.1 Reading the protein data

```
protein <- read.table("protein.txt", sep="\t", header=TRUE)
summary(protein)
          Country      RedMeat         WhiteMeat          Eggs
 Albania        : 1   Min.   : 4.400   Min.   : 1.400   Min.   :0.500
 Austria        : 1   1st Qu.: 7.800   1st Qu.: 4.900   1st Qu.:2.700
 Belgium        : 1   Median : 9.500   Median : 7.800   Median :2.900
 Bulgaria       : 1   Mean   : 9.828   Mean   : 7.896   Mean   :2.936
 Czechoslovakia: 1   3rd Qu.:10.600   3rd Qu.:10.800   3rd Qu.:3.700
 Denmark        : 1   Max.   :18.000   Max.   :14.000   Max.   :4.700
 (Other)        :19
      Milk             Fish           Cereals          Starch
 Min.   : 4.90   Min.   : 0.200   Min.   :18.60   Min.   :0.600
 1st Qu.:11.10   1st Qu.: 2.100   1st Qu.:24.30   1st Qu.:3.100
 Median :17.60   Median : 3.400   Median :28.00   Median :4.700
 Mean   :17.11   Mean   : 4.284   Mean   :32.25   Mean   :4.276
 3rd Qu.:23.30   3rd Qu.: 5.800   3rd Qu.:40.10   3rd Qu.:5.700
 Max.   :33.70   Max.   :14.200   Max.   :56.70   Max.   :6.500

      Nuts            Fr.Veg
 Min.   :0.700   Min.   :1.400
 1st Qu.:1.500   1st Qu.:2.900
 Median :2.400   Median :3.800
 Mean   :3.072   Mean   :4.136
 3rd Qu.:4.700   3rd Qu.:4.900
 Max.   :7.800   Max.   :7.900
```

UNITS AND SCALING

The documentation for this dataset doesn't mention what the units of measurement are, though we can assume all the columns are measured in the same units. This is important: units (or more precisely, disparity in units) affect what clusterings an algorithm will discover. If you measure vital statistics of your subjects as age in years, height in feet, and weight in pounds, you'll get different distances—and possibly different clusters—than if you measure age in years, height in meters, and weight in kilograms.

Ideally, you want a unit of change in each coordinate to represent the same degree of difference. In the `protein` dataset, we assume that the measurements are all in the same units, so it might seem that we're okay. This may well be a correct assumption, but different food groups provide different amounts of protein. Animal-based food sources in general have more grams of protein per serving than plant-based food sources, so one could argue that a change in consumption of 5 grams is a bigger difference in terms of vegetable consumption than it is in terms of red meat consumption.

One way to try to make the clustering more coordinate-free is to transform all the columns to have a mean value of 0 and a standard deviation of 1. This makes the standard deviation the unit of measurement in each coordinate. Assuming that your training data has a distribution that accurately represents the population at large, then a standard deviation represents approximately the same degree of difference in every coordinate. You can scale the data in R using the function `scale()`.

Listing 8.2 Rescaling the dataset

The output of scale() is a matrix. For the purposes of this chapter, you can think of a matrix as a data frame with all numeric columns (this isn't strictly true, but it's close enough).

Use all the columns except the first (Country).

```
vars.to.use <- colnames(protein)[-1]
pmatrix <- scale(protein[,vars.to.use])

pcenter <- attr(pmatrix, "scaled:center")

pscale <- attr(pmatrix, "scaled:scale")
```

The scale() function annotates its output with two attributes—scaled:center returns the mean values of all the columns, and scaled:scale returns the standard deviations. You'll store these away so you can "unscale" the data later.

Now on to clustering. We'll start with hierarchical.

8.1.3 Hierarchical clustering with hclust()

The `hclust()` function takes as input a distance matrix (as an object of class `dist`), which records the distances between all pairs of points in the data (using any one of a variety of metrics). It returns a *dendrogram*: a tree that represents the nested clusters. `hclust()` uses one of a variety of clustering methods to produce a tree that records the nested cluster structure. You can compute the distance matrix using the function `dist()`.

`dist()` will calculate distance functions using the (squared) Euclidean distance (`method="euclidean"`), the Manhattan distance (`method="manhattan"`), and something like the Hamming distance, when categorical variables are expanded to indicators (`method="binary"`). If you want to use another distance metric, you'll have to compute the appropriate distance matrix and convert it to a `dist` object using the `as.dist()` call (see `help(dist)` for further details).

Let's cluster the protein data. We'll use Ward's method, which starts out with each data point as an individual cluster and merges clusters iteratively so as to minimize the *total within sum of squares (WSS)* of the clustering (we'll explain more about WSS later in the chapter).

Listing 8.3 Hierarchical clustering

```
d <- dist(pmatrix, method="euclidean")
pfit <- hclust(d, method="ward")
plot(pfit, labels=protein$Country)
```

Create the distance matrix.

Do the clustering.

Plot the dendrogram.

The dendrogram suggests five clusters (as shown in figure 8.2). You can draw the rectangles on the dendrogram using the function `rect.hclust()`:

```
rect.hclust(pfit, k=5)
```

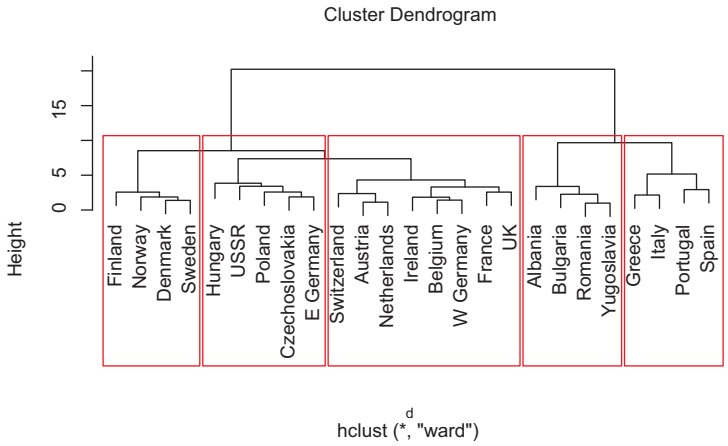

Figure 8.2 **Dendrogram of countries clustered by protein consumption**

To extract the members of each cluster from the `hclust` object, use `cutree()`.

Listing 8.4 Extracting the clusters found by `hclust()`

```
groups <- cutree(pfit, k=5)

print_clusters <- function(labels, k) {
  for(i in 1:k) {
    print(paste("cluster", i))
    print(protein[labels==i,c("Country","RedMeat","Fish","Fr.Veg")])
  }
}
```

A convenience function for printing out the countries in each cluster, along with the values for red meat, fish, and fruit/vegetable consumption. We'll use this function throughout this section. Note that the function is hardcoded for the protein dataset.

```
> print_clusters(groups, 5)
[1] "cluster 1"
      Country RedMeat Fish Fr.Veg
1     Albania    10.1  0.2    1.7
4    Bulgaria     7.8  1.2    4.2
18    Romania     6.2  1.0    2.8
25 Yugoslavia     4.4  0.6    3.2
[1] "cluster 2"
       Country RedMeat Fish Fr.Veg
2      Austria     8.9  2.1    4.3
3      Belgium    13.5  4.5    4.0
9       France    18.0  5.7    6.5
12     Ireland    13.9  2.2    2.9
14 Netherlands     9.5  2.5    3.7
21 Switzerland    13.1  2.3    4.9
22          UK    17.4  4.3    3.3
24   W Germany    11.4  3.4    3.8
[1] "cluster 3"
          Country RedMeat Fish Fr.Veg
5  Czechoslovakia     9.7  2.0    4.0
7       E Germany     8.4  5.4    3.6
11        Hungary     5.3  0.3    4.2
```

```
16          Poland     6.9  3.0    6.6
23            USSR     9.3  3.0    2.9
[1] "cluster 4"
   Country RedMeat Fish Fr.Veg
6  Denmark    10.6  9.9    2.4
8  Finland     9.5  5.8    1.4
15  Norway     9.4  9.7    2.7
20  Sweden     9.9  7.5    2.0
[1] "cluster 5"
    Country RedMeat Fish Fr.Veg
10   Greece    10.2  5.9    6.5
13    Italy     9.0  3.4    6.7
17 Portugal     6.2 14.2    7.9
19    Spain     7.1  7.0    7.2
```

There's a certain logic to these clusters: the countries in each cluster tend to be in the same geographical region. It makes sense that countries in the same region would have similar dietary habits. You can also see that

- Cluster 2 is made of countries with higher-than-average red meat consumption.
- Cluster 4 contains countries with higher-than-average fish consumption but low produce consumption.
- Cluster 5 contains countries with high fish and produce consumption.

This dataset has only 25 points; it's harder to "eyeball" the clusters and the cluster members when there are very many data points. In the next few sections, we'll look at some ways to examine clusters more holistically.

VISUALIZING CLUSTERS

As we mentioned in chapter 3, visualization is an effective way to get an overall view of the data, or in this case, the clusters. We can try to visualize the clustering by projecting the data onto the first two *principal components* of the data.[2] If *N* is the number of variables that describe the data, then the principal components describe the hyperellipsoid in *N*-space that bounds the data. If you order the principal components by the length of the hyperellipsoid's corresponding axes (longest first), then the first two principal components describe a plane in *N*-space that captures as much of the variation of the data as can be captured in two dimensions. We'll use the `prcomp()` call to do the principal components decomposition.

Listing 8.5 Projecting the clusters on the first two principal components

```
library(ggplot2)
princ <- prcomp(pmatrix)
nComp <- 2
project <- predict(princ, newdata=pmatrix)[,1:nComp]
```

Calculate the principal components of the data.

The predict() function will rotate the data into the space described by the principal components. We only want the projection on the first two axes.

[2] We can project the data onto any two of the principal components, but the first two are the most likely to show useful information.

```
project.plus <- cbind(as.data.frame(project),
                      cluster=as.factor(groups),
                      country=protein$Country)
ggplot(project.plus, aes(x=PC1, y=PC2)) +
  geom_point(aes(shape=cluster)) +
  geom_text(aes(label=country),
            hjust=0, vjust=1)
```

Create a data frame with the transformed data, along with the cluster label and country label of each point.

Plot it.

You can see in figure 8.3 that the Romania/Yugoslavia/Bulgaria/Albania cluster and the Mediterranean cluster (Spain and so on) are separated from the others. The other three clusters co-mingle in this projection, though they're probably more separated in other projections.

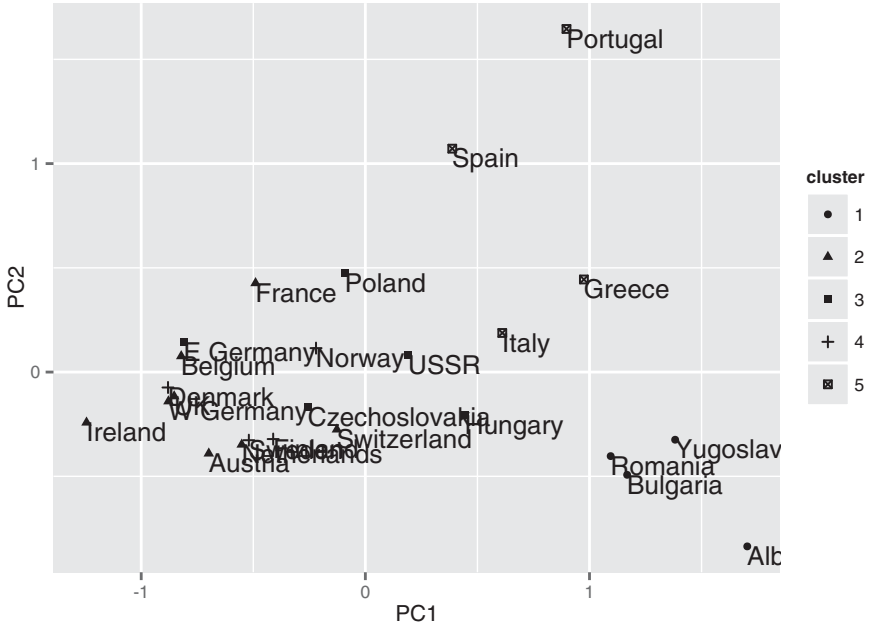

Figure 8.3 Plot of countries clustered by protein consumption, projected onto first two principal components

BOOTSTRAP EVALUATION OF CLUSTERS

An important question when evaluating clusters is whether a given cluster is "real"—does the cluster represent actual structure in the data, or is it an artifact of the clustering algorithm? As you'll see, this is especially important with clustering algorithms like k-means, where the user has to specify the number of clusters *a priori*. It's been our experience that clustering algorithms will often produce several clusters that represent actual structure or relationships in the data, and then one or two clusters that are buckets that represent "other" or "miscellaneous." Clusters of "other" tend to be made up of data points that have no real relationship to each other; they just don't fit anywhere else.

One way to assess whether a cluster represents true structure is to see if the cluster holds up under plausible variations in the dataset. The `fpc` package has a function called `clusterboot()` that uses bootstrap resampling to evaluate how stable a given cluster is.[3] `clusterboot()` is an integrated function that both performs the clustering and evaluates the final produced clusters. It has interfaces to a number of R clustering algorithms, including both `hclust` and `kmeans`.

`clusterboot`'s algorithm uses the *Jaccard coefficient*, a similarity measure between sets. The Jaccard similarity between two sets A and B is the ratio of the number of elements in the intersection of A and B over the number of elements in the union of A and B. The basic general strategy is as follows:

1. Cluster the data as usual.
2. Draw a new dataset (of the same size as the original) by resampling the original dataset with replacement (meaning that some of the data points may show up more than once, and others not at all). Cluster the new dataset.
3. For every cluster in the original clustering, find the most similar cluster in the new clustering (the one that gives the maximum Jaccard coefficient) and record that value. If this maximum Jaccard coefficient is less than 0.5, the original cluster is considered to be *dissolved*—it didn't show up in the new clustering. A cluster that's dissolved too often is probably not a "real" cluster.
4. Repeat steps 2–3 several times.

The cluster stability of each cluster in the original clustering is the mean value of its Jaccard coefficient over all the bootstrap iterations. As a rule of thumb, clusters with a stability value less than 0.6 should be considered unstable. Values between 0.6 and 0.75 indicate that the cluster is measuring a pattern in the data, but there isn't high certainty about which points should be clustered together. Clusters with stability values above about 0.85 can be considered highly stable (they're likely to be real clusters).

Different clustering algorithms can give different stability values, even when the algorithms produce highly similar clusterings, so `clusterboot()` is also measuring how stable the clustering algorithm is.

Let's run `clusterboot()` on the protein data, using hierarchical clustering with five clusters.

Listing 8.6 Running `clusterboot()` on the protein data

```
library(fpc)

kbest.p<-5
```

Load the fpc package. You may have to install it first. We'll discuss installing R packages in appendix A.

Set the desired number of clusters.

[3] For a full description of the algorithm, see Christian Henning, "Cluster-wise assessment of cluster stability," Research Report 271, Dept. of Statistical Science, University College London, December 2006. The report can be found online at http://mng.bz/3XzA.

```
cboot.hclust <- clusterboot(pmatrix,clustermethod=hclustCBI,
                            method="ward", k=kbest.p)
```

Run clusterboot() with hclust ('clustermethod=hclustCBI') using Ward's method ('method="ward"') and kbest.p clusters ('k=kbest.p'). Return the results in an object called cboot.hclust.

```
> summary(cboot.hclust$result)
              Length Class  Mode
result         7     hclust list
noise          1     -none- logical
nc             1     -none- numeric
clusterlist    5     -none- list
partition     25     -none- numeric
clustermethod  1     -none- character
nccl           1     -none- numeric
```

The results of the clustering are in cboot.hclust$result. The output of the hclust() function is in cboot.hclust$result$result.

```
> groups<-cboot.hclust$result$partition
> print_clusters(groups, kbest.p)
[1] "cluster 1"
      Country RedMeat Fish Fr.Veg
1     Albania    10.1  0.2    1.7
4    Bulgaria     7.8  1.2    4.2
18    Romania     6.2  1.0    2.8
25 Yugoslavia     4.4  0.6    3.2
[1] "cluster 2"
       Country RedMeat Fish Fr.Veg
2      Austria     8.9  2.1    4.3
3      Belgium    13.5  4.5    4.0
9       France    18.0  5.7    6.5
12     Ireland    13.9  2.2    2.9
14 Netherlands     9.5  2.5    3.7
21 Switzerland    13.1  2.3    4.9
22          UK    17.4  4.3    3.3
24   W Germany    11.4  3.4    3.8
[1] "cluster 3"
          Country RedMeat Fish Fr.Veg
5  Czechoslovakia     9.7  2.0    4.0
7       E Germany     8.4  5.4    3.6
11        Hungary     5.3  0.3    4.2
16         Poland     6.9  3.0    6.6
23           USSR     9.3  3.0    2.9
[1] "cluster 4"
   Country RedMeat Fish Fr.Veg
6  Denmark    10.6  9.9    2.4
8  Finland     9.5  5.8    1.4
15  Norway     9.4  9.7    2.7
20  Sweden     9.9  7.5    2.0
[1] "cluster 5"
    Country RedMeat Fish Fr.Veg
10   Greece    10.2  5.9    6.5
13    Italy     9.0  3.4    6.7
17 Portugal     6.2 14.2    7.9
19    Spain     7.1  7.0    7.2
```

cboot.hclust$result$partition returns a vector of cluster labels.

The clusters are the same as those produced by a direct call to hclust().

```
> cboot.hclust$bootmean
```

The vector of cluster stabilities.

```
 [1] 0.7905000 0.7990913 0.6173056 0.9312857 0.7560000
> cboot.hclust$bootbrd
 [1] 25 11 47  8 35
```

The count of how many times each cluster was dissolved. By default clusterboot() runs 100 bootstrap iterations.

The `clusterboot()` results show that the cluster of countries with high fish consumption (cluster 4) is highly stable. Clusters 1 and 2 are also quite stable; cluster 5 less so (you can see in figure 8.4 that the members of cluster 5 are separated from the other countries, but also fairly separated from each other). Cluster 3 has the characteristics of what we've been calling the "other" cluster.

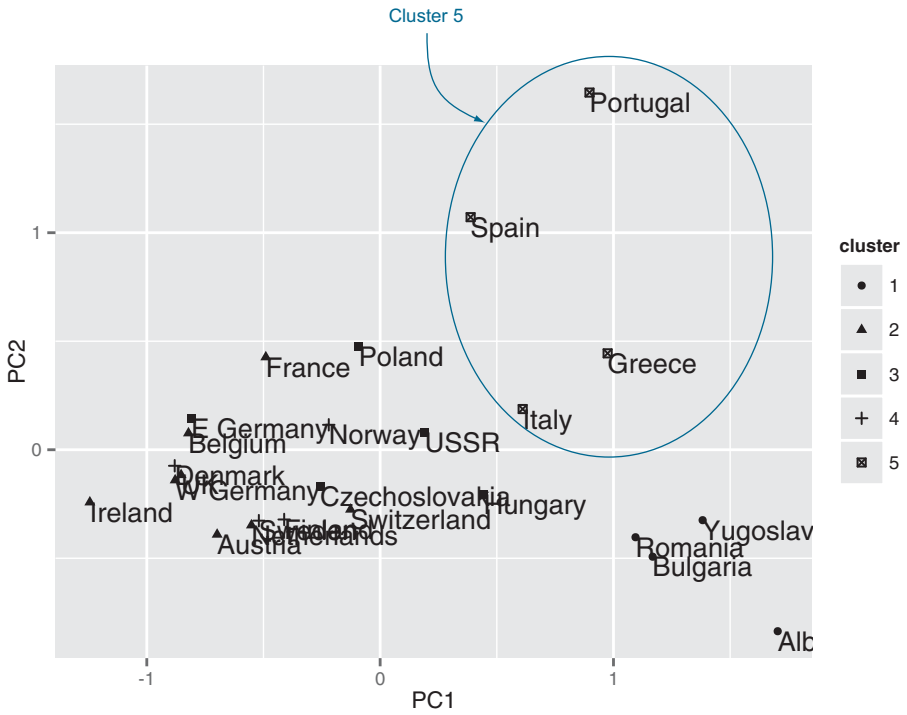

Figure 8.4 Cluster 5: The Mediterranean cluster. Its members are separated from the other clusters, but also from each other.

`clusterboot()` assumes that you know the number of clusters, *k*. We eyeballed the appropriate *k* from the dendrogram, but this isn't always feasible with a large dataset. Can we pick a plausible *k* in a more automated fashion? We'll look at this question in the next section.

PICKING THE NUMBER OF CLUSTERS

There are a number of heuristics and rules-of-thumb for picking clusters; a given heuristic will work better on some datasets than others. It's best to take advantage of

domain knowledge to help set the number of clusters, if that's possible. Otherwise, try a variety of heuristics, and perhaps a few different values of *k*.

Total within sum of squares

One simple heuristic is to compute the *total within sum of squares (WSS)* for different values of *k* and look for an "elbow" in the curve. Define the cluster's *centroid* as the point that is the mean value of all the points in the cluster. The within sum of squares for a single cluster is the average squared distance of each point in the cluster from the cluster's centroid. The total within sum of squares is the sum of the within sum of squares of all the clusters. We show the calculation in the following listing.

Listing 8.7 Calculating total within sum of squares

```
sqr_edist <- function(x, y) {
  sum((x-y)^2)
}

wss.cluster <- function(clustermat) {
  c0 <- apply(clustermat, 2, FUN=mean)
  sum(apply(clustermat, 1, FUN=function(row){sqr_edist(row,c0)}))
}

wss.total <- function(dmatrix, labels) {
  wsstot <- 0
  k <- length(unique(labels))
  for(i in 1:k)
    wsstot <- wsstot + wss.cluster(subset(dmatrix, labels==i))
  wsstot
}
```

Function to calculate squared distance between two vectors.

Function to calculate the WSS for a single cluster, which is represented as a matrix (one row for every point).

Calculate the centroid of the cluster (the mean of all the points).

Calculate the squared difference of every point in the cluster from the centroid, and sum all the distances.

Function to compute the total WSS from a set of data points and cluster labels.

Extract each cluster, calculate the cluster's WSS, and sum all the values.

The total WSS will decrease as the number of clusters increases, because each cluster will be smaller and tighter. The hope is that the rate at which the WSS decreases will slow down for *k* beyond the optimal number of clusters. In other words, the graph of WSS versus *k* should flatten out beyond the optimal *k*, so the optimal *k* will be at the "elbow" of the graph. Unfortunately, this elbow can be difficult to see.

Calinski-Harabasz index

The *Calinski-Harabasz index* of a clustering is the ratio of the between-cluster variance (which is essentially the variance of all the cluster centroids from the dataset's grand centroid) to the total within-cluster variance (basically, the average WSS of the clusters in the clustering). For a given dataset, the *total sum of squares (TSS)* is the squared distance of all the data points from the dataset's centroid. The TSS is independent of the clustering. If WSS(k) is the total WSS of a clustering with k clusters, then the *between sum of squares* BSS(k) of the clustering is given by `BSS(k) = TSS - WSS(k)`. WSS(k) measures how close the points in a cluster are to each other. BSS(k) measures how far

apart the clusters are from each other. A good clustering has a small WSS(k) and a large BSS(k).

The within-cluster variance W is given by `WSS(k)/(n-k)`, where n is the number of points in the dataset. The between-cluster variance B is given by `BSS(k)/(k-1)`. The within-cluster variance will decrease as k increases; the rate of decrease should slow down past the optimal k. The between-cluster variance will increase as k, but the rate of increase should slow down past the optimal k. So in theory, the ratio of B to W should be maximized at the optimal k.

Let's write a function to calculate the Calinski-Harabasz (CH) index. The function will accommodate both a `kmeans` clustering and an `hclust` clustering.

Listing 8.8 The Calinski-Harabasz index

```
totss <- function(dmatrix) {                                  # Convenience function to calculate the total sum of squares.
  grandmean <- apply(dmatrix, 2, FUN=mean)
  sum(apply(dmatrix, 1, FUN=function(row){sqr_edist(row, grandmean)}))
}

                                                              # A function to calculate the CH index for a number of clusters from 1 to kmax.
ch_criterion <- function(dmatrix, kmax, method="kmeans") {
  if(!(method %in% c("kmeans", "hclust"))) {
    stop("method must be one of c('kmeans', 'hclust')")
  }
  npts <- dim(dmatrix)[1]  # number of rows.


  totss <- totss(dmatrix)                                     # The total sum of squares is independent of the clustering.


  wss <- numeric(kmax)
  crit <- numeric(kmax)
  wss[1] <- (npts-1)*sum(apply(dmatrix, 2, var))              # Calculate WSS for k=1 (which is really just total sum of squares).
  for(k in 2:kmax) {                                          # Calculate WSS for k from 2 to kmax. kmeans() returns the total WSS as one of its outputs.
    if(method=="kmeans") {
      clustering<-kmeans(dmatrix, k, nstart=10, iter.max=100)
      wss[k] <- clustering$tot.withinss
    }else {  # hclust                                         # For hclust(), calculate total WSS by hand.
      d <- dist(dmatrix, method="euclidean")
      pfit <- hclust(d, method="ward")
      labels <- cutree(pfit, k=k)
      wss[k] <- wss.total(dmatrix, labels)
    }
  }
```

```
  bss <- totss - wss                          <- Calculate BSS for k from 1 to kmax.
  crit.num <- bss/(0:(kmax-1))                <- Normalize BSS by k-1.
  crit.denom <- wss/(npts - 1:kmax)           <- Normalize WSS by npts - k.
  list(crit = crit.num/crit.denom, wss = wss, totss = totss)
}
```

Return a vector of CH indices and of WSS for k from 1 to kmax. Also return total sum of squares.

We can calculate both indices for the `protein` dataset and plot them.

Listing 8.9 Evaluating clusterings with different numbers of clusters

```
library(reshape2)                                  # Load the reshape2 package (for the melt() function).

clustcrit <- ch_criterion(pmatrix, 10, method="hclust")   # Calculate both criteria for 1–10 clusters.
critframe <- data.frame(k=1:10, ch=scale(clustcrit$crit),   # Create a data frame with the number of clusters, the CH criterion, and the WSS criterion. We'll scale both the CH and WSS criteria to similar ranges so that we can plot them both on the same graph.
                        wss=scale(clustcrit$wss))
critframe <- melt(critframe, id.vars=c("k"),       # Use the melt() function to put the data frame in a shape suitable for ggplot.
                  variable.name="measure",
                  value.name="score")

ggplot(critframe, aes(x=k, y=score, color=measure)) +   # Plot it.
  geom_point(aes(shape=measure)) + geom_line(aes(linetype=measure)) +
  scale_x_continuous(breaks=1:10, labels=1:10)
```

Looking at figure 8.5, you see that the CH criterion is maximized at `k=2`, with another local maximum at `k=5`. If you squint your eyes, you can convince yourself that the WSS plot has an elbow at `k=2`. The `k=2` clustering corresponds to the first split of the dendrogram in figure 8.2; if you use `clusterboot()` to do the clustering, you'll see that the clusters are highly stable, though perhaps not very informative.

There are several other indices that you can try when picking *k*. The *gap statistic*[4] is an attempt to automate the "elbow finding" on the WSS curve. It works best when the data comes from a mix of populations that all have approximately Gaussian distributions (a *mixture of Gaussian*). We'll see one more measure, the *average silhouette width*, when we discuss `kmeans()`.

[4] See Robert Tibshirani, Guenther Walther, and Trevor Hastie, "Estimating the number of clusters in a data set via the gap statistic," *Journal of the Royal Statistical Society B*, 2001, 63(2), pp. 411-423; www.stanford.edu/~hastie/Papers/gap.pdf.

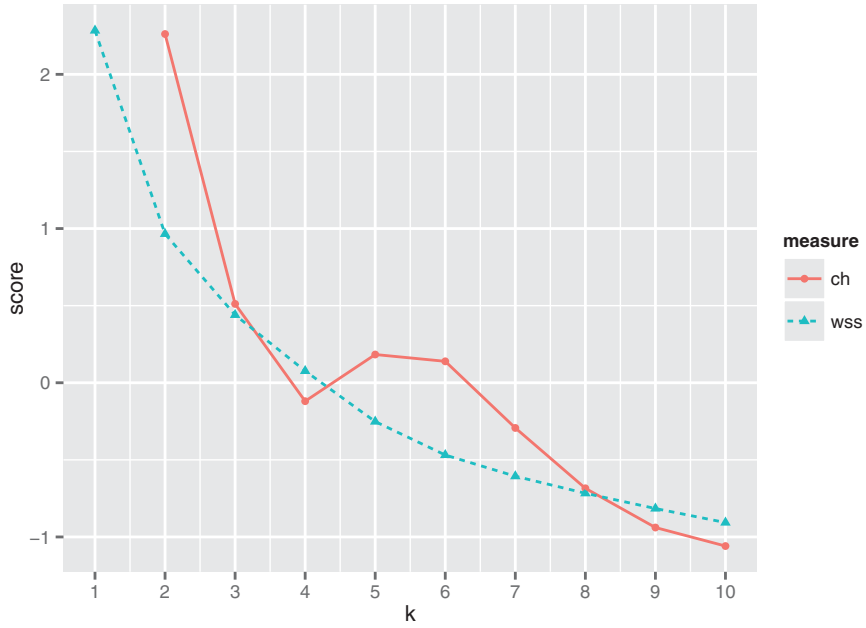

Figure 8.5 Plot of the Calinski-Harabasz and WSS indices for 1–10 clusters, on protein data

8.1.4 The k-means algorithm

K-means is a popular clustering algorithm when the data is all numeric and the distance metric is squared Euclidean (though you could in theory run it with other distance metrics). It's fairly ad hoc and has the major disadvantage that you must pick k in advance. On the plus side, it's easy to implement (one reason it's so popular) and can be faster than hierarchical clustering on large datasets. It works best on data that looks like a mixture of Gaussians (which the `protein` data unfortunately doesn't appear to be).

THE KMEANS() FUNCTION

The function to run k-means in R is `kmeans()`. The output of `kmeans()` includes the cluster labels, the centers (centroids) of the clusters, the total sum of squares, total WSS, total BSS, and the WSS of each cluster. The k-means algorithm is illustrated in figure 8.6, with $k = 2$.

This algorithm isn't guaranteed to have a unique stopping point. K-means can be fairly unstable, in that the final clusters depend on the initial cluster centers. It's good practice to run k-means several times with different random starts, and then select the clustering with the lowest total WSS. The `kmeans()` function can do this automatically, though it defaults to only using one random start.

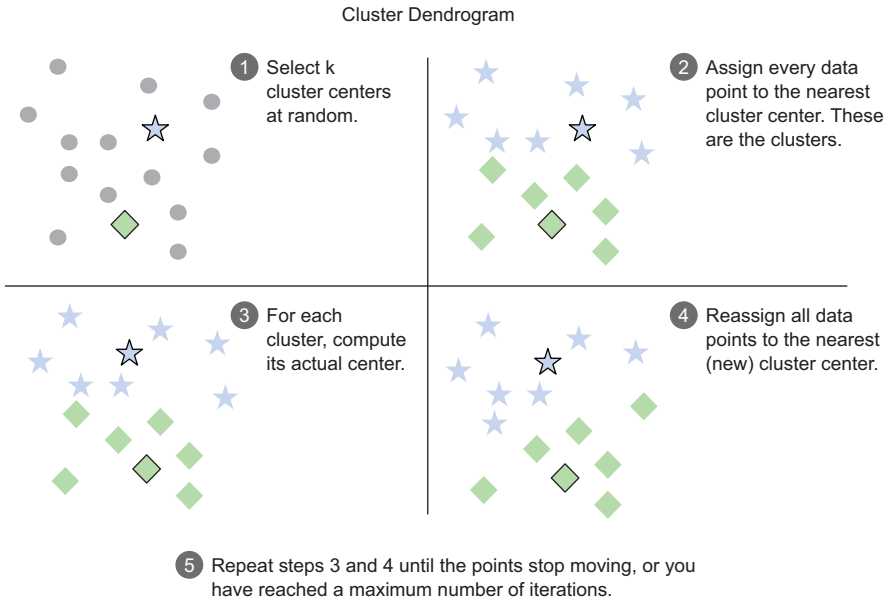

Figure 8.6 The k-means procedure. The two cluster centers are represented by the outlined star and diamond.

Let's run `kmeans()` on the `protein` data (scaled to 0 mean and unit standard deviation, as before). We'll use `k=5`, as shown in the next listing.

Listing 8.10 Running k-means with `k=5`

```
> pclusters <- kmeans(pmatrix, kbest.p, nstart=100, iter.max=100)
 > summary(pclusters)
             Length Class  Mode
cluster      25     -none- numeric
centers      45     -none- numeric
totss         1     -none- numeric
withinss      5     -none- numeric
tot.withinss  1     -none- numeric
betweenss     1     -none- numeric
size          5     -none- numeric

> pclusters$centers
       RedMeat  WhiteMeat        Eggs       Milk        Fish
1 -0.807569986 -0.8719354 -1.55330561 -1.0783324 -1.0386379
2  0.006572897 -0.2290150  0.19147892  1.3458748  1.1582546
3 -0.570049402  0.5803879 -0.08589708 -0.4604938 -0.4537795
4  1.011180399  0.7421332  0.94084150  0.5700581 -0.2671539
5 -0.508801956 -1.1088009 -0.41248496 -0.8320414  0.9819154
     Cereals     Starch       Nuts      Fr.Veg
1  1.7200335 -1.4234267  0.9961313 -0.64360439
2 -0.8722721  0.1676780 -0.9553392 -1.11480485
3  0.3181839  0.7857609 -0.2679180  0.06873983
```

kmeans() returns all the sum of squares measures.

Run kmeans() with five clusters (kbest.p=5), 100 random starts, and 100 maximum iterations per run.

pclusters$centers is a matrix whose rows are the centroids of the clusters. Note that pclusters$centers is in the scaled coordinates, not the original protein coordinates.

```
4 -0.6877583  0.2288743 -0.5083895  0.02161979
5  0.1300253 -0.1842010  1.3108846  1.62924487
> pclusters$size
[1] 4 4 5 8 4
```

pclusters$size returns the number of points in each cluster. Generally (though not always) a good clustering will be fairly well balanced: no extremely small clusters and no extremely large ones.

pclusters$cluster is a vector of cluster labels.

```
> groups <- pclusters$cluster
 > print_clusters(groups, kbest.p)
 [1] "cluster 1"
      Country RedMeat Fish Fr.Veg
1     Albania    10.1  0.2    1.7
4    Bulgaria     7.8  1.2    4.2
18    Romania     6.2  1.0    2.8
25 Yugoslavia     4.4  0.6    3.2
[1] "cluster 2"
   Country RedMeat Fish Fr.Veg
6  Denmark    10.6  9.9    2.4
8  Finland     9.5  5.8    1.4
15  Norway     9.4  9.7    2.7
20  Sweden     9.9  7.5    2.0
[1] "cluster 3"
          Country RedMeat Fish Fr.Veg
5  Czechoslovakia     9.7  2.0    4.0
7       E Germany     8.4  5.4    3.6
11        Hungary     5.3  0.3    4.2
16         Poland     6.9  3.0    6.6
23           USSR     9.3  3.0    2.9
[1] "cluster 4"
       Country RedMeat Fish Fr.Veg
2      Austria     8.9  2.1    4.3
3      Belgium    13.5  4.5    4.0
9       France    18.0  5.7    6.5
12     Ireland    13.9  2.2    2.9
14 Netherlands     9.5  2.5    3.7
21 Switzerland    13.1  2.3    4.9
22          UK    17.4  4.3    3.3
24   W Germany    11.4  3.4    3.8
[1] "cluster 5"
    Country RedMeat Fish Fr.Veg
10   Greece    10.2  5.9    6.5
13    Italy     9.0  3.4    6.7
17 Portugal     6.2 14.2    7.9
19    Spain     7.1  7.0    7.2
```

In this case, kmeans() and hclust() returned the same clustering. This won't always be true.

THE KMEANSRUNS() FUNCTION FOR PICKING K

To run `kmeans()`, you must know *k*. The `fpc` package (the same package that has `clusterboot()`) has a function called `kmeansruns()` that calls `kmeans()` over a range of *k* and estimates the best *k*. It then returns its pick for the best value of *k*, the output of `kmeans()` for that value, and a vector of criterion values as a function of *k*. Currently, `kmeansruns()` has two criteria: the *Calinski-Harabasz Index* (`"ch"`), and the *average silhouette width* (`"asw"`; for more about silhouette clustering, see http://mng.bz/Qe15). It's a good idea to plot the criterion values over the entire range of *k*, since you may see evidence for a *k* that the algorithm didn't automatically pick (as we did in figure 8.5), as we demonstrate in the following listing.

Listing 8.11 Plotting cluster criteria

Run kmeansruns() from 1–10 clusters, and the CH criterion. By default, kmeansruns() uses 100 random starts and 100 maximum iterations per run.

```
> clustering.ch <- kmeansruns(pmatrix, krange=1:10, criterion="ch")
> clustering.ch$bestk
[1] 2
```

The CH criterion picks two clusters.

```
> clustering.asw <- kmeansruns(pmatrix, krange=1:10, criterion="asw")
> clustering.asw$bestk
[1] 3
```

Run kmeansruns() from 1–10 clusters, and the average silhouette width criterion. Average silhouette width picks 3 clusters.

The vector of criterion values is called crit.

```
> clustering.ch$crit
 [1]  0.000000 14.094814 11.417985 10.418801 10.011797  9.964967
 [7]  9.861682  9.412089  9.166676  9.075569
> clustcrit$crit
 [1]       NaN 12.215107 10.359587  9.690891 10.011797  9.964967
 [7]  9.506978  9.092065  8.822406  8.695065
```

Compare the CH values for kmeans() and hclust(). They're not quite the same, because the two algorithms didn't pick the same clusters.

Plot the values for the two criteria.

```
> critframe <- data.frame(k=1:10, ch=scale(clustering.ch$crit),
            asw=scale(clustering.asw$crit))
> critframe <- melt(critframe, id.vars=c("k"),
                variable.name="measure",
                 value.name="score")
> ggplot(critframe, aes(x=k, y=score, color=measure)) +
   geom_point(aes(shape=measure)) + geom_line(aes(linetype=measure)) +
   scale_x_continuous(breaks=1:10, labels=1:10)
> summary(clustering.ch)
             Length Class  Mode
cluster      25     -none- numeric
centers      18     -none- numeric
totss         1     -none- numeric
withinss      2     -none- numeric
tot.withinss  1     -none- numeric
betweenss     1     -none- numeric
size          2     -none- numeric
crit         10     -none- numeric
bestk         1     -none- numeric
```

kmeansruns() also returns the output of kmeans for k=bestk.

Figure 8.7 shows the results of the two clustering criteria provided by `kmeansruns`. They suggest two to three clusters as the best choice. However, if you compare the values of `clustering.ch$crit` and `clustcrit$crit` in the listing, you'll see that the CH

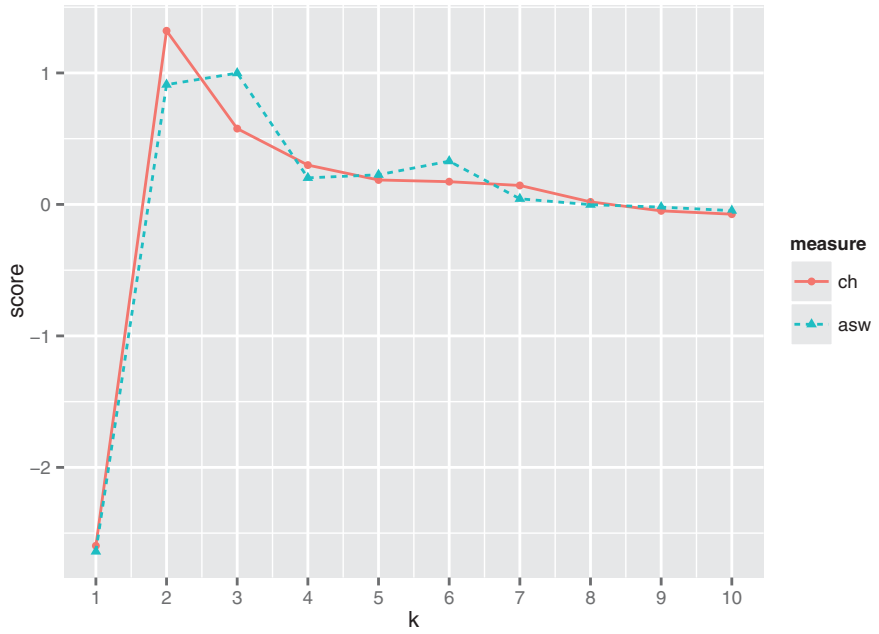

Figure 8.7 Plot of the Calinski-Harabasz and average silhouette width indices for 1–10 clusters, on protein data

criterion produces different curves for `kmeans()` and `hclust()` clusterings, but it did pick the same value (which probably means it picked the same clusters) for `k=5`, and `k=6`, which might be taken as evidence that either five or six is the optimal choice for *k*.

CLUSTERBOOT() REVISITED

We can run `clusterboot()` using the k-means algorithm, as well.

Listing 8.12 Running `clusterboot()` with k-means

```
kbest.p<-5
cboot<-clusterboot(pmatrix, clustermethod=kmeansCBI,
            runs=100,iter.max=100,
            krange=kbest.p, seed=15555)

> groups <- cboot$result$partition
> print_clusters(cboot$result$partition, kbest.p)
[1] "cluster 1"
      Country RedMeat Fish Fr.Veg
1     Albania    10.1  0.2    1.7
4    Bulgaria     7.8  1.2    4.2
18    Romania     6.2  1.0    2.8
25 Yugoslavia     4.4  0.6    3.2
[1] "cluster 2"
   Country RedMeat Fish Fr.Veg
6  Denmark    10.6  9.9    2.4
8  Finland     9.5  5.8    1.4
```

We've set the seed for the random generator so the results are reproducible.

```
15  Norway     9.4  9.7    2.7
20  Sweden     9.9  7.5    2.0
[1] "cluster 3"
          Country RedMeat Fish Fr.Veg
5  Czechoslovakia     9.7  2.0    4.0
7       E Germany     8.4  5.4    3.6
11        Hungary     5.3  0.3    4.2
16         Poland     6.9  3.0    6.6
23           USSR     9.3  3.0    2.9
[1] "cluster 4"
       Country RedMeat Fish Fr.Veg
2      Austria     8.9  2.1    4.3
3      Belgium    13.5  4.5    4.0
9       France    18.0  5.7    6.5
12     Ireland    13.9  2.2    2.9
14 Netherlands     9.5  2.5    3.7
21 Switzerland    13.1  2.3    4.9
22          UK    17.4  4.3    3.3
24   W Germany    11.4  3.4    3.8
[1] "cluster 5"
    Country RedMeat Fish Fr.Veg
10   Greece    10.2  5.9    6.5
13    Italy     9.0  3.4    6.7
17 Portugal     6.2 14.2    7.9
19    Spain     7.1  7.0    7.2
> cboot$bootmean
[1] 0.8670000 0.8420714 0.6147024 0.7647341 0.7508333
> cboot$bootbrd
[1] 15 20 49 17 32
```

Note that the stability numbers as given by `cboot$bootmean` (and the number of times that the clusters were "dissolved" as given by `cboot$bootbrd`) are different for the hierarchical clustering and k-means, even though the discovered clusters are the same. This shows that the stability of a clustering is partly a function of the clustering algorithm, not just the data. Again, the fact that both clustering algorithms discovered the same clusters might be taken as an indication that five is the optimal number of clusters.

8.1.5 Assigning new points to clusters

Clustering is often used as part of data exploration, or as a precursor to other supervised learning methods. But you may want to use the clusters that you discovered to categorize new data, as well. One common way to do so is to treat the centroid of each cluster as the representative of the cluster as a whole, and then assign new points to the cluster with the nearest centroid. Note that if you scaled the original data before clustering, then you should also scale the new data point the same way before assigning it to a cluster.

Listing 8.13 A function to assign points to a cluster

```
assign_cluster <- function(newpt, centers, xcenter=0, xscale=1) {
  xpt <- (newpt - xcenter)/xscale
  dists <- apply(centers, 1, FUN=function(c0){sqr_edist(c0, xpt)})
  which.min(dists)
}
```

A function to assign a new data point newpt to a clustering described by centers, a matrix where each row is a cluster centroid. If the data was scaled (using scale()) before clustering, then xcenter and xscale are the scaled:center and scaled:scale attributes, respectively.

Center and scale the new data point.

Calculate how far the new data point is from each of the cluster centers.

Return the cluster number of the closest centroid.

Note that the function `sqr_edist` (the squared Euclidean distance) was defined previously, in section 8.1.1.

Let's look at an example of assigning points to clusters, using synthetic data.

Listing 8.14 An example of assigning points to clusters

```
rnorm.multidim <- function(n, mean, sd, colstr="x") {
   ndim <- length(mean)
   data <- NULL
   for(i in 1:ndim) {
     col <- rnorm(n, mean=mean[[i]], sd=sd[[i]])
     data<-cbind(data, col)
   }
   cnames <- paste(colstr, 1:ndim, sep='')
   colnames(data) <- cnames
   data
 }
```

A function to generate n points drawn from a multidimensional Gaussian distribution with centroid mean and standard deviation sd. The dimension of the distribution is given by the length of the vector mean.

```
mean1 <- c(1, 1, 1)
sd1 <- c(1, 2, 1)

mean2 <- c(10, -3, 5)
sd2 <- c(2, 1, 2)

mean3 <- c(-5, -5, -5)
sd3 <- c(1.5, 2, 1)
```

The parameters for three Gaussian distributions.

```
clust1 <- rnorm.multidim(100, mean1, sd1)
clust2 <- rnorm.multidim(100, mean2, sd2)
clust3 <- rnorm.multidim(100, mean3, sd3)
toydata <- rbind(clust3, rbind(clust1, clust2))
```

Create a dataset with 100 points each drawn from the above distributions.

Scale the dataset.

```
tmatrix <- scale(toydata)
tcenter <- attr(tmatrix, "scaled:center")
tscale<-attr(tmatrix, "scaled:scale")
kbest.t <- 3
tclusters <- kmeans(tmatrix, kbest.t, nstart=100, iter.max=100)
```

Store the centering and scaling parameters for future use.

Cluster the dataset, using k-means with three clusters.

```
tclusters$size
[1] 100 101  99
```

The resulting clusters are about the right size.

```
unscale <- function(scaledpt, centervec, scalevec) {
   scaledpt*scalevec + centervec
}
```

A function to "unscale" data points (put them back in the coordinates of the original dataset).

```
> unscale(tclusters$centers[1,], tcenter, tscale)
       x1        x2        x3
 9.978961 -3.097584  4.864689
> mean2
[1] 10 -3  5
```

Unscale the first centroid. It corresponds to our original distribution 2.

```
> unscale(tclusters$centers[2,], tcenter, tscale)
       x1        x2        x3
-4.979523 -4.927404 -4.908949
> mean3
[1] -5 -5 -5
```

The second centroid corresponds to the original distribution 3.

```
> unscale(tclusters$centers[3,], tcenter, tscale)
       x1        x2        x3
1.0003356 1.3037825 0.9571058
> mean1
[1] 1 1 1
```

The third centroid corresponds to the original distribution 1.

```
> assign_cluster(rnorm.multidim(1, mean1, sd1),
                 tclusters$centers,
                 tcenter, tscale)
3
3
```

Generate a random point from the original distribution 1 and assign it to one of the discovered clusters.

It's assigned to cluster 3, as we would expect.

```
> assign_cluster(rnorm.multidim(1, mean2, sd1),
                 tclusters$centers,
                 tcenter, tscale)
1
1
```

Generate a random point from the original distribution 2 and assign it.

It's assigned to cluster 1.

```
> assign_cluster(rnorm.multidim(1, mean3, sd1),
                 tclusters$centers,
                 tcenter, tscale)
2
2
```

Generate a random point from the original distribution 3 and assign it.

It's assigned to cluster 2.

8.1.6 Clustering takeaways

Here's what you should remember about clustering:

- The goal of clustering is to discover or draw out similarities among subsets of your data.
- In a good clustering, points in the same cluster should be more similar (nearer) to each other than they are to points in other clusters.
- When clustering, the units that each variable is measured in matter. Different units cause different distances and potentially different clusterings.
- Ideally, you want a unit change in each coordinate to represent the same degree of change. One way to approximate this is to transform all the columns to have a mean value of 0 and a standard deviation of 1.0, for example by using the function `scale()`.
- Clustering is often used for data exploration or as a precursor to supervised learning methods.
- Like visualization, it's more iterative and interactive, and less automated than supervised methods.
- Different clustering algorithms will give different results. You should consider different approaches, with different numbers of clusters.
- There are many heuristics for estimating the best number of clusters. Again, you should consider the results from different heuristics and explore various numbers of clusters.

Sometimes, rather than looking for subsets of data points that are highly similar to each other, you'd like to know what kind of data (or which data attributes) tend to occur together. In the next section, we'll look at one approach to this problem.

8.2 Association rules

Association rule mining is used to find objects or attributes that frequently occur together—for example, products that are often bought together during a shopping session, or queries that tend to occur together during a session on a website's search engine. Such information can be used to recommend products to shoppers, to place frequently bundled items together on store shelves, or to redesign websites for easier navigation.

8.2.1 Overview of association rules

The unit of "togetherness" when mining association rules is called a *transaction*. Depending on the problem, a transaction could be a single shopping basket, a single user session on a website, or even a single customer. The objects that comprise a transaction are referred to as *items* in an *itemset*: the products in the shopping basket, the pages visited during a website session, the actions of a customer. Sometimes transactions are referred to as *baskets*, from the shopping basket analogy.

Mining for association rules occurs in two steps:

1. Look for all the itemsets (subsets of transactions) that occur more often than in a minimum fraction of the transactions.
2. Turn those itemsets into rules.

Let's consider the example of books that are checked out from a library. When a library patron checks out a set of books, that's a transaction; the books that the patron checked out are the itemset that comprise the transaction. Table 8.1 represents a database of transactions.

Table 8.1 A database of library transactions

Transaction ID	Books checked out
1	*The Hobbit*, *The Princess Bride*
2	*The Princess Bride*, *The Last Unicorn*
3	*The Hobbit*
4	*The Neverending Story*
5	*The Last Unicorn*
6	*The Hobbit*, *The Princess Bride*, *The Fellowship of the Ring*
7	*The Hobbit*, *The Fellowship of the Ring*, *The Two Towers*, *The Return of the King*
8	*The Fellowship of the Ring*, *The Two Towers*, *The Return of the King*
9	*The Hobbit*, *The Princess Bride*, *The Last Unicorn*
10	*The Last Unicorn*, *The Neverending Story*

Looking over all the transactions in table 8.1, you find that *The Hobbit* is in 50% of all transactions, and *The Princess Bride* is in 40% of them (you run a library where fantasy is quite popular). Both books are checked out together in 30% of all transaction. We'd say the *support* of the itemset {*The Hobbit*, *The Princess Bride*} is 30%. Of the five transactions that include *The Hobbit*, three (60%) also include *The Princess Bride*. So you can make a rule "People who check out *The Hobbit* also check out *The Princess Bride*." This rule should be correct (according to your data) 60% of the time. We'd say that the *confidence* of the rule is 60%. Conversely, of the four times *The Princess Bride* was checked

out, *The Hobbit* appeared three times, or 75% of the time. So the rule "People who check out *The Princess Bride* also check out *The Hobbit*" has 75% confidence.

Let's define support and confidence formally. The rule "if X, then Y" means that every time you see the itemset X in a transaction, you expect to also see Y (with a given confidence). For the apriori algorithm (which we'll look at in this section), Y is always an itemset with one item. Suppose that your database of transactions is called T. Then `support(X)` is the number of transactions that contain X divided by the total number of transactions in T. The confidence of the rule "if X, then Y" is given by `conf(X=>Y) = support(union(X,Y))/support(X)`, where `union(X, Y)` means that you're referring to itemsets that contain both the items in X and the items in Y.

The goal in association rule mining is to find all the interesting rules in the database with at least a given minimum support (say, 10%) and a minimum given confidence (say, 60%).

8.2.2 The example problem

For our example problem, let's imagine that we're working for a bookstore, and we want to identify books that our customers are interested in, based on (all of) their previous purchases and book interests. We can get information about their book interests two ways: either they've purchased a book from us, or they've rated the book on our website (even if they bought the book somewhere else). In this case, a transaction is a customer, and an itemset is all the books that they've expressed an interest in, either by purchase or by rating.

The data that we'll use is based on data collected in 2004 from the book community Book-Crossing[5] for research conducted at the Institut für Informatik, University of Freiburg.[6] We've condensed the information into a single tab-separated text file called bookdata.tsv. Each row of the file consists of a user ID, a book title (which we've designed as a unique ID for each book), and the rating (which we won't actually use in this example):

```
"token" "userid"        "rating"        "title"
" a light in the storm" 55927   0       " A Light in the Storm"
```

The `token` column contains lower-cased column strings; we used the tokens to identify books with different ISBNs (the original book IDs) that had the same title except for

[5] The original data repository can be found at http://mng.bz/2052. Since some artifacts in the original files caused errors when reading into R, we're providing copies of the data as a prepared RData object:https://github.com/WinVector/zmPDSwR/blob/master/Bookdata/bxBooks.RData. The prepared version of the data that we'll use in this section is at https://github.com/WinVector/zmPDSwR/blob/master/Bookdata/bookdata.tsv.gz. Further information and scripts for preparing the data can be found at https://github.com/WinVector/zmPDSwR/tree/master/Bookdata.

[6] The researchers' original paper is "Improving Recommendation Lists Through Topic Diversification," Cai-Nicolas Ziegler, Sean M. McNee, Joseph A. Konstan, Georg Lausen; Proceedings of the 14th International World Wide Web Conference (WWW '05), May 10-14, 2005, Chiba, Japan. It can be found online at http://mng.bz/7trR.

casing. The `title` column holds properly capitalized title strings; these are unique per book, so we'll use them as book IDs.

In this format, the transaction (customer) information is diffused through the data, rather than being all in one row; this reflects the way the data would naturally be stored in a database, since the customer's activity would be diffused throughout time. Books generally come in different editions or from different publishers. We've condensed all different versions into a single item; hence different copies or printings of *Little Women* will all map to the same item ID in our data (namely, the title `Little Women`).

The original data includes approximately a million ratings of 271,379 books from 278,858 readers. Our data will have fewer books due to the mapping that we discussed earlier.

Now we're ready to mine.

8.2.3 Mining association rules with the arules package

We'll use the package `arules` for association rule mining. `arules` includes an implementation of the popular association rule algorithm *apriori*, as well as implementations to read in and examine transaction data.[7] The package uses special data types to hold and manipulate the data; we'll explore these data types as we work the example.

READING IN THE DATA

We can read the data directly from the bookdata.tsv.gz file into the object `bookbaskets` using the function `read.transaction()`.

Listing 8.15 Reading in the book data

```
library(arules)                                  # Load the arules package.

bookbaskets <- read.transactions("bookdata.tsv.gz", format="single",   # Specify the file and the file format.
                                 sep="\t",       # Specify the column separator (a tab).
                                 cols=c("userid", "title"),   # Specify the column of transaction IDs and of item IDs, respectively.
                                 rm.duplicates=T)   # Tell the function to look for and remove duplicate entries (for example, multiple entries for The Hobbit by the same user).
```

The `read.transactions()` function reads data in two formats: the format where every row corresponds to a single item (like `bookdata.tsv.gz`), and a format where each row corresponds to a single transaction, possibly with a transaction ID, like table 8.1. To read data in the first format, use the argument `format="single"`; to read data in the second format, use the argument `format="basket"`.

[7] For a more comprehensive introduction to `arules` than we can give in this chapter, please see Hahsler, Grin, Hornik, and Buchta, "Introduction to arules—A computational environment for mining association rules and frequent item sets," online at cran.r-project.org/web/packages/arules/vignettes/arules.pdf.

It sometimes happens that a reader will buy one edition of a book and then later add a rating for that book under a different edition. Because of the way we're representing books for this example, these two actions will result in duplicate entries. The `rm.duplicates=T` argument will eliminate them. It will also output some (not too useful) diagnostics about the duplicates.

Once you've read in the data, you can inspect the resulting object.

EXAMINING THE DATA

Transactions are represented as a special object called `transactions`. You can think of a `transactions` object as a 0/1 matrix, with one row for every transaction and one column for every possible item. The matrix entry *(i,j)* is 1 if the *i* transaction contains item *j*. There are a number of calls you can use to examine the transaction data, as the next listing shows.

Listing 8.16 Examining the transaction data

```
> class(bookbaskets)                          <-- The object is of class transactions.
[1] "transactions"
attr(,"package")
[1] "arules"                                  <-- Printing the object tells you its dimensions.
> bookbaskets
transactions in sparse format with
 92108 transactions (rows) and
 220447 items (columns)
> dim(bookbaskets)                            <-- You can also use dim() to see the dimensions of the matrix.
[1]  92108 220447
> colnames(bookbaskets)[1:5]                  <-- The columns are labeled by book title.
[1] " A Light in the Storm:[...]"
[2] " Always Have Popsicles"
[3] " Apple Magic"
[4] " Ask Lily"
[5] " Beyond IBM: Leadership Marketing and Finance for the 1990s"
> rownames(bookbaskets)[1:5]                  <-- The rows are labeled by customer.
[1] "10"     "1000"   "100001" "100002" "100004"
```

You can examine the distribution of transaction sizes (or basket sizes) with the function `size()`:

```
> basketSizes <- size(bookbaskets)
> summary(basketSizes)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    1.0     1.0     1.0    11.1     4.0 10250.0
```

Most customers (at least half of them, in fact) only expressed interest in one book. But someone has expressed interest in more than 10,000! You probably want to look more closely at the size distribution to see what's going on.

Listing 8.17 Examining the size distribution

```
> quantile(basketSizes, probs=seq(0,1,0.1))
    0%   10%   20%   30%   40%   50%   60%   70%   80%   90%  100%
     1     1     1     1     1     1     2     3     5    13 10253
> library(ggplot2)
 > ggplot(data.frame(count=basketSizes)) +
  geom_density(aes(x=count), binwidth=1) +
  scale_x_log10()
```

Look at the basket size distribution, in 10% increments.

Plot the distribution to get a better look.

Figure 8.8 shows the distribution of basket sizes. 90% of customers expressed interest in fewer than 15 books; most of the remaining customers expressed interest in up to about 100 books or so (the call `quantile(basketSizes, probs=c(0.99, 1))` will show you that 99% of customers expressed interest in 179 books or fewer). Still, there are a few people who have expressed interest in several hundred, or even several thousand books.

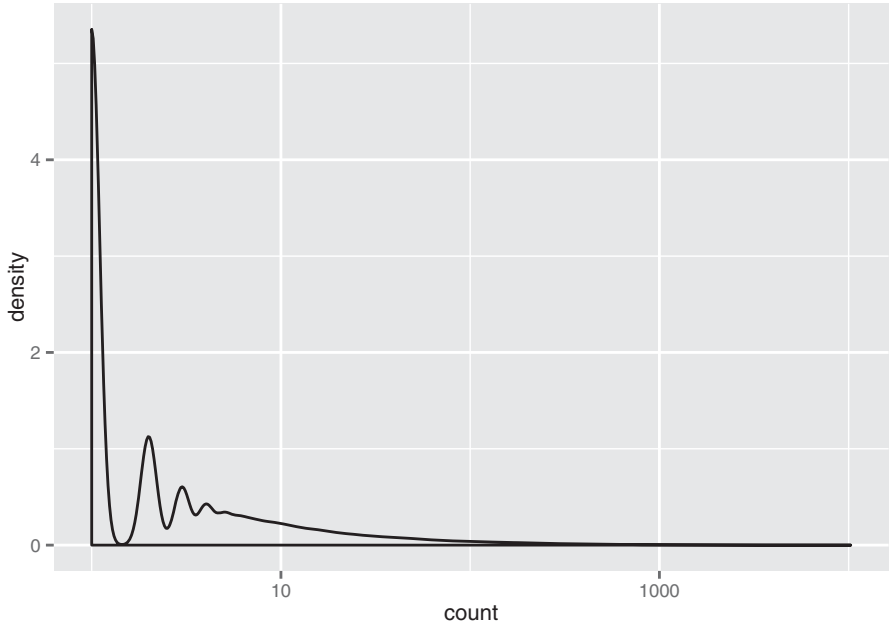

Figure 8.8 A density plot of basket sizes

Which books are they reading? The function `itemFrequency()` will give you the relative frequency of each book in the transaction data:

```
> bookFreq <- itemFrequency(bookbaskets)
summary(bookFreq)
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
1.086e-05 1.086e-05 1.086e-05 5.035e-05 3.257e-05 2.716e-02

> sum(bookFreq)
[1] 11.09909
```

Note that the frequencies don't sum to 1. You can recover the number of times that each book occurred in the data by normalizing the item frequencies and multiplying by the total number of items.

Listing 8.18 Finding the ten most frequent books

```
> bookCount <- (bookFreq/sum(bookFreq))*sum(basketSizes)
 > summary(bookCount)
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
   1.000    1.000    1.000    4.637    3.000 2502.000
> orderedBooks <- sort(bookCount, decreasing=T)
 > orderedBooks[1:10]
                                    Wild Animus
                                           2502
                      The Lovely Bones: A Novel
                                           1295
                             She's Come Undone
                                            934
                             The Da Vinci Code
                                            905
          Harry Potter and the Sorcerer's Stone
                                            832
                    The Nanny Diaries: A Novel
                                            821
                                 A Painted House
                                            819
                         Bridget Jones's Diary
                                            772
                       The Secret Life of Bees
                                            762
Divine Secrets of the Ya-Ya Sisterhood: A Novel
                                            737
> orderedBooks[1]/dim(bookbaskets)[1]
Wild Animus
 0.02716376
```

Get the absolute count of book occurrences.

Sort the count and list the 10 most popular books.

The most popular book in the dataset occurred in fewer than 3% of the baskets.

The last observation in the preceding listing highlights one of the issues with mining high-dimensional data: when you have thousands of variables, or thousands of items, almost every event is rare. Keep this point in mind when deciding on support thresholds for rule mining; your thresholds will often need to be quite low.

Before we get to the rule mining, let's refine the data a bit more. As we observed earlier, half of the customers in the data only expressed interest in a single book. Since you want to find books that occur together in people's interest lists, you can't make any direct use of people who haven't yet shown interest in multiple books. You can restrict the dataset to customers who have expressed interest in at least two books:

```
> bookbaskets_use <- bookbaskets[basketSizes > 1]
> dim(bookbaskets_use)
[1]  40822 220447
```

Now you're ready to look for association rules.

THE APRIORI() FUNCTION

In order to mine rules, you need to decide on a minimum support level and a minimum threshold level. For this example, let's try restricting the itemsets that we'll consider to those that are supported by at least 100 people. This leads to a minimum support of `100/dim(bookbaskets_use)[1] = 100/40822`. This is about 0.002, or 0.2%. We'll use a confidence threshold of 75%.

Listing 8.19 Finding the association rules

Call apriori() with a minimum support of 0.002 and a minimum confidence of 0.75.

```
> rules <- apriori(bookbaskets_use,
                   parameter =list(support = 0.002, confidence=0.75))
```

The summary of the apriori() output reports the number of rules found;...

```
> summary(rules)
set of 191 rules
```

...the distribution of rule lengths (in this example, most rules contain 3 items—2 on the left side, X (lhs), and one on the right side, Y (rhs));...

```
rule length distribution (lhs + rhs):sizes
  2   3   4   5
 11 100  66  14

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.000   3.000   3.000   3.435   4.000   5.000
```

...a summary of rule quality measures, including support and confidence;...

```
summary of quality measures:
    support           confidence          lift
 Min.   :0.002009   Min.   :0.7500   Min.   : 40.89
 1st Qu.:0.002131   1st Qu.:0.8113   1st Qu.: 86.44
 Median :0.002278   Median :0.8468   Median :131.36
 Mean   :0.002593   Mean   :0.8569   Mean   :129.68
 3rd Qu.:0.002695   3rd Qu.:0.9065   3rd Qu.:158.77
 Max.   :0.005830   Max.   :0.9882   Max.   :321.89
```

...and some information on how apriori() was called.

```
mining info:
            data ntransactions support confidence
 bookbaskets_use          40822   0.002       0.75
```

The quality measures on the rules include not only the rules' support and confidence, but also a quantity called *lift*. Lift compares the frequency of an observed pattern with how often you'd expect to see that pattern just by chance. The lift of a rule "if X, then Y" is given by `support(union(X, Y))/(support(X)*support(Y))`. If the lift is near 1, then there's a good chance that the pattern you observed is occurring just by chance. The larger the lift, the more likely that the pattern is "real." In this case, all the discovered rules have a lift of at least 40, so they're likely to be real patterns in customer behavior.

INSPECTING AND EVALUATING RULES

There are also other metrics and interest measures you can use to evaluate the rules by using the function `interestMeasure()`. We'll look at two of these measures: `coverage` and `fishersExactTest`. *Coverage* is the support of the left side of the rule (X); it tells

you how often the rule would be applied in the dataset. *Fisher's exact test* is a significance test for whether an observed pattern is real, or chance (the same thing lift measures; Fisher's test is more formal). Fisher's exact test returns the p-value, or the probability that you would see the observed pattern by chance; you want the p-value to be small.

Listing 8.20 Scoring rules

```
> measures <- interestMeasure(rules,
+                  method=c("coverage", "fishersExactTest"),
+                  transactions=bookbaskets_use)
> summary(measures)
    coverage         fishersExactTest
 Min.   :0.002082   Min.   : 0.000e+00
 1st Qu.:0.002511   1st Qu.: 0.000e+00
 Median :0.002719   Median : 0.000e+00
 Mean   :0.003039   Mean   :5.080e-138
 3rd Qu.:0.003160   3rd Qu.: 0.000e+00
 Max.   :0.006982   Max.   :9.702e-136
```

The call to interestMeasure() takes as arguments the discovered rules,...

...a list of interest measures to apply,...

...and a dataset to evaluate the interest measures over. This is usually the same set used to mine the rules, but it needn't be. For instance, you can evaluate the rules over the full dataset, bookbaskets, to get coverage estimates that reflect all the customers, not just the ones who showed interest in more than one book.

The coverage of the discovered rules ranges from 0.002–0.007, equivalent to a range of about 100–250 people. All the p-values from Fisher's test are small, so it's likely that the rules reflect actual customer behavior patterns.

You can also call `interestMeasure()` with methods `support`, `confidence`, and `lift`, among others. This would be useful in our example if you wanted to get support, confidence, and lift estimates for the full dataset `bookbaskets`, rather than the filtered dataset `bookbaskets_use`—or for a subset of the data, for instance, only customers from the United States.

The function `inspect()` pretty-prints the rules. The function `sort()` allows you to sort the rules by a quality or interest measure, like confidence. To print the five most confident rules in the dataset, you could use the following command:

```
inspect(head((sort(rules, by="confidence")), n=5))
```

For legibility, we show the output of this command in table 8.2.

Table 8.2 The five most confident rules discovered in the data

Left side	Right side	Support	Confidence	Lift
Four to Score *High Five* *Seven Up* *Two for the Dough*	*Three to Get Deadly*	0.002	0.988	165

Table 8.2 The five most confident rules discovered in the data *(continued)*

Left side	Right side	Support	Confidence	Lift
Harry Potter and the Order of the Phoenix *Harry Potter and the Prisoner of Azkaban* *Harry Potter and the Sorcerer's Stone*	*Harry Potter and the Chamber of Secrets*	0.003	0.966	73
Four to Score *High Five* *One for the Money* *Two for the Dough*	*Three to Get Deadly*	0.002	0.966	162
Four to Score *Seven Up* *Three to Get Deadly* *Two for the Dough*	*High Five*	0.002	0.966	181
High Five *Seven Up* *Three to Get Deadly* *Two for the Dough*	*Four to Score*	0.002	0.966	168

There are two things to notice in table 8.2. First, the rules concern books that come in series: the numbered series of novels about bounty hunter Stephanie Plum, and the Harry Potter series. So these rules essentially say that if a reader has read four Stephanie Plum or Harry Potter books, they're almost sure to buy another one.

The second thing to notice is that rules 1, 4, and 5 are permutations of the same itemset. This is likely to happen when the rules get long.

RESTRICTING WHICH ITEMS TO MINE

You can restrict which items appear in the left side or right side of a rule. Suppose you're interested specifically in books that tend to co-occur with the novel *The Lovely Bones*. You can do this by restricting which books appear on the right side of the rule, using the `appearance` parameter.

Listing 8.21 Finding rules with restrictions

```
brules <- apriori(bookbaskets_use,
                  parameter =list(support = 0.001,
                                  confidence=0.6),
                  appearance=list(rhs=c("The Lovely Bones: A Novel"),
                                  default="lhs"))
> summary(brules)
set of 46 rules
```

Relax the minimum support to 0.001 and the minimum confidence to 0.6.

Only The Lovely Bones is allowed to appear on the right side of the rules.

By default, all the books can go into the left side of the rules.

```
rule length distribution (lhs + rhs):sizes
 3  4
44  2

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  3.000   3.000   3.000   3.043   3.000   4.000

summary of quality measures:
    support           confidence          lift
 Min.   :0.001004   Min.   :0.6000   Min.   :21.81
 1st Qu.:0.001029   1st Qu.:0.6118   1st Qu.:22.24
 Median :0.001102   Median :0.6258   Median :22.75
 Mean   :0.001132   Mean   :0.6365   Mean   :23.14
 3rd Qu.:0.001219   3rd Qu.:0.6457   3rd Qu.:23.47
 Max.   :0.001396   Max.   :0.7455   Max.   :27.10

mining info:
            data ntransactions support confidence
 bookbaskets_use         40822   0.001        0.6
```

The supports, confidences, and lifts are lower than they were in our previous example, but the lifts are still much greater than 1, so it's likely that the rules reflect real customer behavior patterns.

Let's inspect the rules, sorted by confidence. Since they'll all have the same right side, you can use the `lhs()` function to only look at the left sides.

Listing 8.22 Inspecting rules

```
brulesConf <- sort(brules, by="confidence")
```

Sort the rules by confidence.

```
> inspect(head(lhs(brulesConf), n=5))
   items
1 {Divine Secrets of the Ya-Ya Sisterhood: A Novel,
   Lucky : A Memoir}
2 {Lucky : A Memoir,
   The Notebook}
3 {Lucky : A Memoir,
   Wild Animus}
4 {Midwives: A Novel,
   Wicked: The Life and Times of the Wicked Witch of the West}
5 {Lucky : A Memoir,
   Summer Sisters}
```

Use the lhs() function to get the left itemsets of each rule; then inspect the top five.

Note that four of the five most confident rules include *Lucky: A Memoir* in the left side, which perhaps isn't surprising, since *Lucky* was written by the author of *The Lovely Bones*. Suppose you want to find out about works by other authors that are interesting to people who showed interest in *The Lovely Bones*; you can use `subset()` to filter down to only rules that don't include *Lucky*.

Listing 8.23 Inspecting rules with restrictions

```
brulesSub <- subset(brules, subset=!(lhs %in% "Lucky : A Memoir"))
brulesConf <- sort(brulesSub, by="confidence")

> inspect(head(lhs(brulesConf), n=5))
  items
1 {Midwives: A Novel,
   Wicked: The Life and Times of the Wicked Witch of the West}
2 {She's Come Undone,
   The Secret Life of Bees,
   Wild Animus}
3 {A Walk to Remember,
   The Nanny Diaries: A Novel}
4 {Beloved,
   The Red Tent}
5 {The Da Vinci Code,
   The Reader}
```

Restrict to the subset of rules where Lucky is not in the left side.

These examples show that association rule mining is often highly interactive. To get interesting rules, you must often set the support and confidence levels fairly low; as a result you can get many, many rules. Some rules will be more interesting or surprising to you than others; to find them requires sorting the rules by different interest measures, or perhaps restricting yourself to specific subsets of rules.

8.2.4 Association rule takeaways

Here's what you should remember about association rules:

- The goal of association rule mining is to find relationships in the data: items or attributes that tend to occur together.
- A good rule "if X, then Y" should occur more often than you'd expect to observe by chance. You can use lift or Fisher's exact test to check if this is true.
- When a large number of different possible items can be in a basket (in our example, thousands of different books), most events will be rare (have low support).
- Association rule mining is often interactive, as there can be many rules to sort and sift through.

8.3 Summary

In this chapter, you've learned how to find similarities in data using two different clustering methods in R, and how to find items that tend to occur together in data using association rules. You've also learned how to evaluate your discovered clusters and your discovered rules.

Unsupervised methods like the ones we've covered in this chapter are really more exploratory in nature. Unlike with supervised methods, there's no "ground truth" to evaluate your findings against. But the findings from unsupervised methods can be the starting point for more focused experiments and modeling.

In the last few chapters, we've covered the most basic modeling and data analysis techniques; they're all good first approaches to consider when you're starting a new project. In the next chapter, we'll touch on a few more advanced methods.

Key takeaways

- Unsupervised methods find structure in the data, often as a prelude to predictive modeling.
- The goal of clustering is to discover or draw out similarities among subsets of your data.
- When clustering, you'll find that scaling is important.
- The goal of association rule mining is to find relationships in the data: items or attributes that tend to occur together.
- In association rule mining, most events will be rare, so support and confidence levels must often be set low.