CHAPTER 7
Unsupervised Learning

The term *unsupervised learning* refers to statistical methods that extract meaning from data without training a model on labeled data (data where an outcome of interest is known). In Chapters 4 to 6, the goal is to build a model (set of rules) to predict a response variable from a set of predictor variables. This is supervised learning. In contrast, unsupervised learning also constructs a model of the data, but it does not distinguish between a response variable and predictor variables.

Unsupervised learning can be used to achieve different goals. In some cases, it can be used to create a predictive rule in the absence of a labeled response. *Clustering* methods can be used to identify meaningful groups of data. For example, using the web clicks and demographic data of a user on a website, we may be able to group together different types of users. The website could then be personalized to these different types.

In other cases, the goal may be to *reduce the dimension* of the data to a more manageable set of variables. This reduced set could then be used as input into a predictive model, such as regression or classification. For example, we may have thousands of sensors to monitor an industrial process. By reducing the data to a smaller set of features, we may be able to build a more powerful and interpretable model to predict process failure than could be built by including data streams from thousands of sensors.

Finally, unsupervised learning can be viewed as an extension of the exploratory data analysis (see Chapter 1) to situations in which you are confronted with a large number of variables and records. The aim is to gain insight into a set of data and how the different variables relate to each other. Unsupervised techniques allow you to sift through and analyze these variables and discover relationships.

Unsupervised Learning and Prediction

Unsupervised learning can play an important role in prediction, both for regression and classification problems. In some cases, we want to predict a category in the absence of any labeled data. For example, we might want to predict the type of vegetation in an area from a set of satellite sensory data. Since we don't have a response variable to train a model, clustering gives us a way to identify common patterns and categorize the regions.

Clustering is an especially important tool for the "cold-start problem." In this type of problem, such as launching a new marketing campaign or identifying potential new types of fraud or spam, we initially may not have any response to train a model. Over time, as data is collected, we can learn more about the system and build a traditional predictive model. But clustering helps us start the learning process more quickly by identifying population segments.

Unsupervised learning is also important as a building block for regression and classification techniques. With big data, if a small subpopulation is not well represented in the overall population, the trained model may not perform well for that subpopulation. With clustering, it is possible to identify and label subpopulations. Separate models can then be fit to the different subpopulations. Alternatively, the subpopulation can be represented with its own feature, forcing the overall model to explicitly consider subpopulation identity as a predictor.

Principal Components Analysis

Often, variables will vary together (covary), and some of the variation in one is actually duplicated by variation in another (e.g., restaurant checks and tips). Principal components analysis (PCA) is a technique to discover the way in which numeric variables covary.[1]

1 This and subsequent sections in this chapter © 2020 Datastats, LLC, Peter Bruce, Andrew Bruce, and Peter Gedeck; used with permission.

Key Terms for Principal Components Analysis

Principal component
A linear combination of the predictor variables.

Loadings
The weights that transform the predictors into the components.

Synonym
Weights

Screeplot
A plot of the variances of the components, showing the relative importance of the components, either as explained variance or as proportion of explained variance.

The idea in PCA is to combine multiple numeric predictor variables into a smaller set of variables, which are weighted linear combinations of the original set. The smaller set of variables, the *principal components*, "explains" most of the variability of the full set of variables, reducing the dimension of the data. The weights used to form the principal components reveal the relative contributions of the original variables to the new principal components.

PCA was first proposed by Karl Pearson. In what was perhaps the first paper on unsupervised learning, Pearson recognized that in many problems there is variability in the predictor variables, so he developed PCA as a technique to model this variability. PCA can be viewed as the unsupervised version of linear discriminant analysis; see"Discriminant Analysis" on page 201.

A Simple Example

For two variables, X_1 and X_2, there are two principal components Z_i (i = 1 or 2):

$$Z_i = w_{i,1}X_1 + w_{i,2}X_2$$

The weights $\left(w_{i,1}, w_{i,2}\right)$ are known as the component *loadings*. These transform the original variables into the principal components. The first principal component, Z_1, is the linear combination that best explains the total variation. The second principal component, Z_2, is orthogonal to the first and explains as much of the remaining variation as it can. (If there were additional components, each additional one would be orthogonal to the others.)

It is also common to compute principal components on deviations from the means of the predictor variables, rather than on the values themselves.

You can compute principal components in *R* using the `princomp` function. The following performs a PCA on the stock price returns for Chevron (CVX) and ExxonMobil (XOM):

```
oil_px <- sp500_px[, c('CVX', 'XOM')]
pca <- princomp(oil_px)
pca$loadings
```

```
Loadings:
    Comp.1 Comp.2
CVX -0.747  0.665
XOM -0.665 -0.747

               Comp.1 Comp.2
SS loadings       1.0    1.0
Proportion Var    0.5    0.5
Cumulative Var    0.5    1.0
```

In *Python*, we can use the `scikit-learn` implementation `sklearn.decomposition.PCA`:

```
pcs = PCA(n_components=2)
pcs.fit(oil_px)
loadings = pd.DataFrame(pcs.components_, columns=oil_px.columns)
loadings
```

The weights for CVX and XOM for the first principal component are –0.747 and –0.665, and for the second principal component they are 0.665 and –0.747. How to interpret this? The first principal component is essentially an average of CVX and XOM, reflecting the correlation between the two energy companies. The second principal component measures when the stock prices of CVX and XOM diverge.

It is instructive to plot the principal components with the data. Here we create a visualization in *R*:

```
loadings <- pca$loadings
ggplot(data=oil_px, aes(x=CVX, y=XOM)) +
  geom_point(alpha=.3) +
  stat_ellipse(type='norm', level=.99) +
  geom_abline(intercept = 0, slope = loadings[2,1]/loadings[1,1]) +
  geom_abline(intercept = 0, slope = loadings[2,2]/loadings[1,2])
```

The following code creates a similar visualization in *Python*:

```
def abline(slope, intercept, ax):
    """Calculate coordinates of a line based on slope and intercept"""
    x_vals = np.array(ax.get_xlim())
    return (x_vals, intercept + slope * x_vals)

ax = oil_px.plot.scatter(x='XOM', y='CVX', alpha=0.3, figsize=(4, 4))
ax.set_xlim(-3, 3)
ax.set_ylim(-3, 3)
ax.plot(*abline(loadings.loc[0, 'CVX'] / loadings.loc[0, 'XOM'], 0, ax),
        '--', color='C1')
ax.plot(*abline(loadings.loc[1, 'CVX'] / loadings.loc[1, 'XOM'], 0, ax),
        '--', color='C1')
```

The result is shown in Figure 7-1.

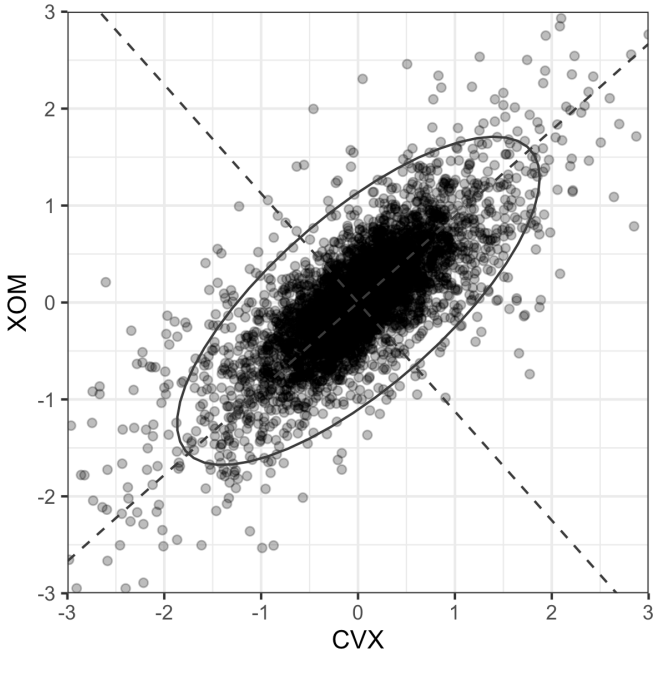

Figure 7-1. The principal components for the stock returns for Chevron (CVX) and ExxonMobil (XOM)

The dashed lines show the direction of the two principal components: the first one is along the long axis of the ellipse, and the second one is along the short axis. You can see that a majority of the variability in the two stock returns is explained by the first principal component. This makes sense since energy stock prices tend to move as a group.

The weights for the first principal component are both negative, but reversing the sign of all the weights does not change the principal component. For example, using weights of 0.747 and 0.665 for the first principal component is equivalent to the negative weights, just as an infinite line defined by the origin and 1,1 is the same as one defined by the origin and –1, –1.

Computing the Principal Components

Going from two variables to more variables is straightforward. For the first component, simply include the additional predictor variables in the linear combination, assigning weights that optimize the collection of the covariation from all the predictor variables into this first principal component (*covariance* is the statistical term; see "Covariance Matrix" on page 202). Calculation of principal components is a classic statistical method, relying on either the correlation matrix of the data or the covariance matrix, and it executes rapidly, not relying on iteration. As noted earlier, principal components analysis works only with numeric variables, not categorical ones. The full process can be described as follows:

1. In creating the first principal component, PCA arrives at the linear combination of predictor variables that maximizes the percent of total variance explained.
2. This linear combination then becomes the first "new" predictor, Z_1.
3. PCA repeats this process, using the same variables with different weights, to create a second new predictor, Z_2. The weighting is done such that Z_1 and Z_2 are uncorrelated.
4. The process continues until you have as many new variables, or components, Z_i as original variables X_i.
5. Choose to retain as many components as are needed to account for most of the variance.
6. The result so far is a set of weights for each component. The final step is to convert the original data into new principal component scores by applying the weights to the original values. These new scores can then be used as the reduced set of predictor variables.

Interpreting Principal Components

The nature of the principal components often reveals information about the structure of the data. There are a couple of standard visualization displays to help you glean insight about the principal components. One such method is a *screeplot* to visualize the relative importance of principal components (the name derives from the resemblance of the plot to a scree slope; here, the y-axis is the eigenvalue). The following *R* code shows an example for a few top companies in the S&P 500:

```
syms <- c( 'AAPL', 'MSFT', 'CSCO', 'INTC', 'CVX', 'XOM',
   'SLB', 'COP', 'JPM', 'WFC', 'USB', 'AXP', 'WMT', 'TGT', 'HD', 'COST')
top_sp <- sp500_px[row.names(sp500_px)>='2005-01-01', syms]
sp_pca <- princomp(top_sp)
screeplot(sp_pca)
```

The information to create a loading plot from the `scikit-learn` result is available in `explained_variance_`. Here, we convert it into a `pandas` data frame and use it to make a bar chart:

```
syms = sorted(['AAPL', 'MSFT', 'CSCO', 'INTC', 'CVX', 'XOM', 'SLB', 'COP',
               'JPM', 'WFC', 'USB', 'AXP', 'WMT', 'TGT', 'HD', 'COST'])
top_sp = sp500_px.loc[sp500_px.index >= '2011-01-01', syms]

sp_pca = PCA()
sp_pca.fit(top_sp)

explained_variance = pd.DataFrame(sp_pca.explained_variance_)
ax = explained_variance.head(10).plot.bar(legend=False, figsize=(4, 4))
ax.set_xlabel('Component')
```

As seen in Figure 7-2, the variance of the first principal component is quite large (as is often the case), but the other top principal components are significant.

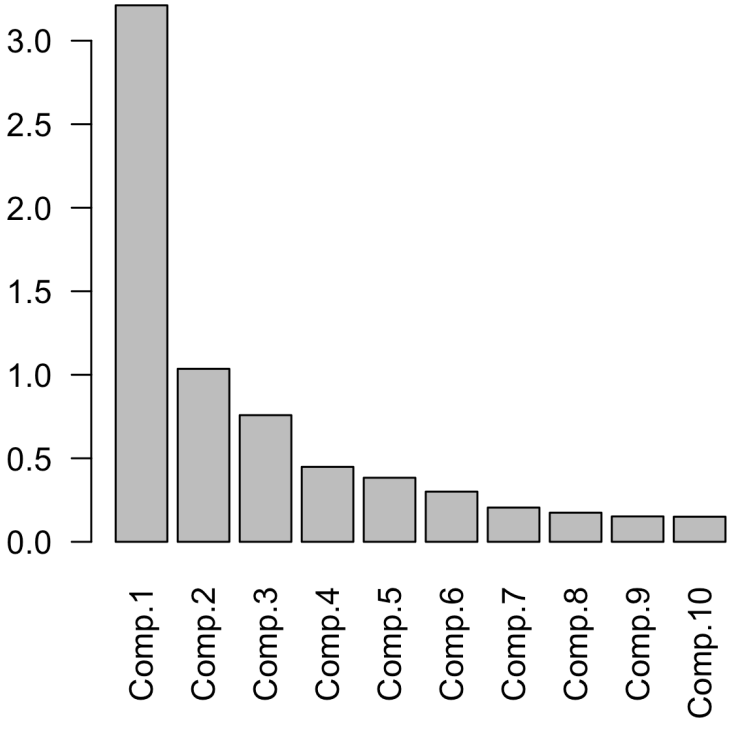

Figure 7-2. A screeplot for a PCA of top stocks from the S&P 500

It can be especially revealing to plot the weights of the top principal components. One way to do this in *R* is to use the `gather` function from the `tidyr` package in conjunction with `ggplot`:

```
library(tidyr)
loadings <- sp_pca$loadings[,1:5]
loadings$Symbol <- row.names(loadings)
loadings <- gather(loadings, 'Component', 'Weight', -Symbol)
ggplot(loadings, aes(x=Symbol, y=Weight)) +
  geom_bar(stat='identity') +
  facet_grid(Component ~ ., scales='free_y')
```

Here is the code to create the same visualization in *Python*:

```
loadings = pd.DataFrame(sp_pca.components_[0:5, :], columns=top_sp.columns)
maxPC = 1.01 * np.max(np.max(np.abs(loadings.loc[0:5, :])))

f, axes = plt.subplots(5, 1, figsize=(5, 5), sharex=True)
for i, ax in enumerate(axes):
    pc_loadings = loadings.loc[i, :]
    colors = ['C0' if l > 0 else 'C1' for l in pc_loadings]
    ax.axhline(color='#888888')
```

```
    pc_loadings.plot.bar(ax=ax, color=colors)
    ax.set_ylabel(f'PC{i+1}')
    ax.set_ylim(-maxPC, maxPC)
```

The loadings for the top five components are shown in Figure 7-3. The loadings for the first principal component have the same sign: this is typical for data in which all the columns share a common factor (in this case, the overall stock market trend). The second component captures the price changes of energy stocks as compared to the other stocks. The third component is primarily a contrast in the movements of Apple and CostCo. The fourth component contrasts the movements of Schlumberger (SLB) to the other energy stocks. Finally, the fifth component is mostly dominated by financial companies.

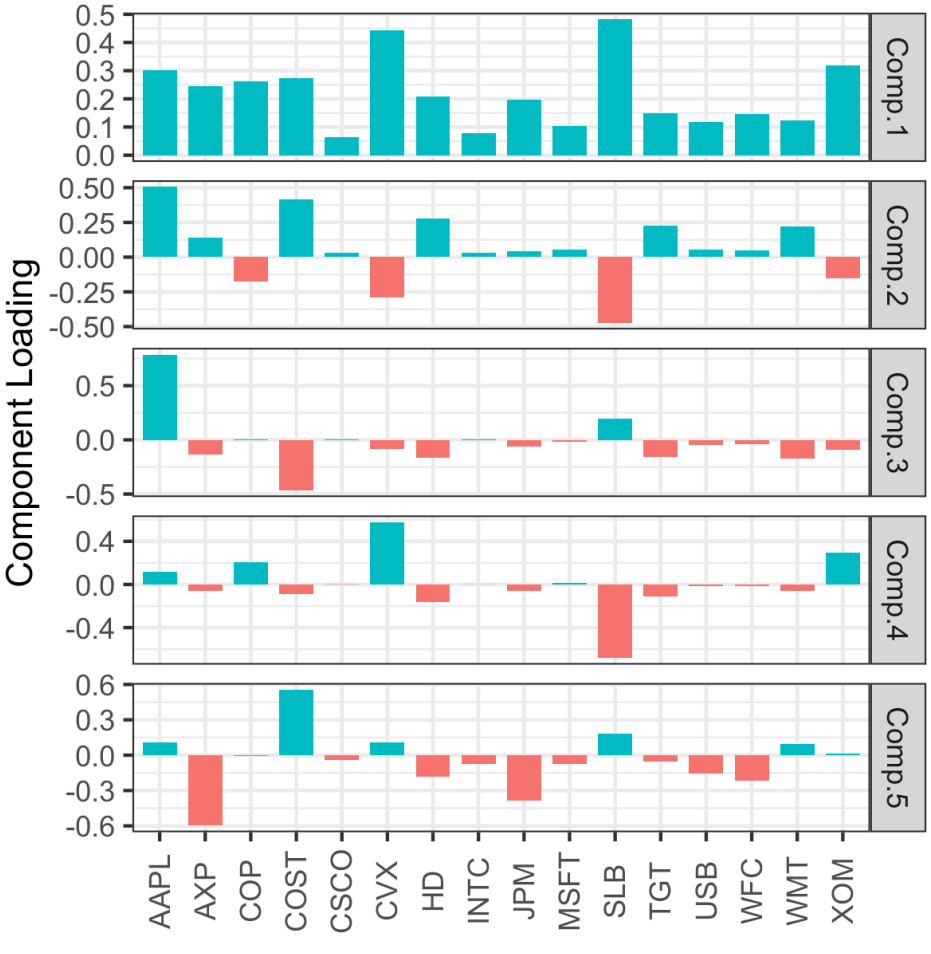

Figure 7-3. The loadings for the top five principal components of stock price returns

How Many Components to Choose?

If your goal is to reduce the dimension of the data, you must decide how many principal components to select. The most common approach is to use an ad hoc rule to select the components that explain "most" of the variance. You can do this visually through the screeplot, as, for example, in Figure 7-2. Alternatively, you could select the top components such that the cumulative variance exceeds a threshold, such as 80%. Also, you can inspect the loadings to determine if the component has an intuitive interpretation. Cross-validation provides a more formal method to select the number of significant components (see "Cross-Validation" on page 155 for more).

Correspondence Analysis

PCA cannot be used for categorical data; however, a somewhat related technique is *correspondence analysis*. The goal is to recognize associations between categories, or between categorical features. The similarities between correspondence analysis and principal components analysis are mainly under the hood—the matrix algebra for dimension scaling. Correspondence analysis is used mainly for graphical analysis of low-dimensional categorical data and is not used in the same way that PCA is for dimension reduction as a preparatory step with big data.

The input can be seen as a table, with rows representing one variable and columns another, and the cells representing record counts. The output (after some matrix algebra) is a *biplot*—a scatterplot with axes scaled (and with percentages indicating how much variance is explained by that dimension). The meaning of the units on the axes is not intuitively connected to the original data, and the main value of the scatterplot is to illustrate graphically variables that are associated with one another (by proximity on the plot). See for example, Figure 7-4, in which household tasks are arrayed according to whether they are done jointly or solo (vertical axis), and whether wife or husband has primary responsibility (horizontal axis). Correspondence analysis is many decades old, as is the spirit of this example, judging by the assignment of tasks.

There are a variety of packages for correspondence analysis in *R*. Here, we use the package `ca`:

```
ca_analysis <- ca(housetasks)
plot(ca_analysis)
```

In *Python*, we can use the `prince` package, which implements correspondence analysis using the `scikit-learn` API:

```
ca = prince.CA(n_components=2)
ca = ca.fit(housetasks)

ca.plot_coordinates(housetasks, figsize=(6, 6))
```

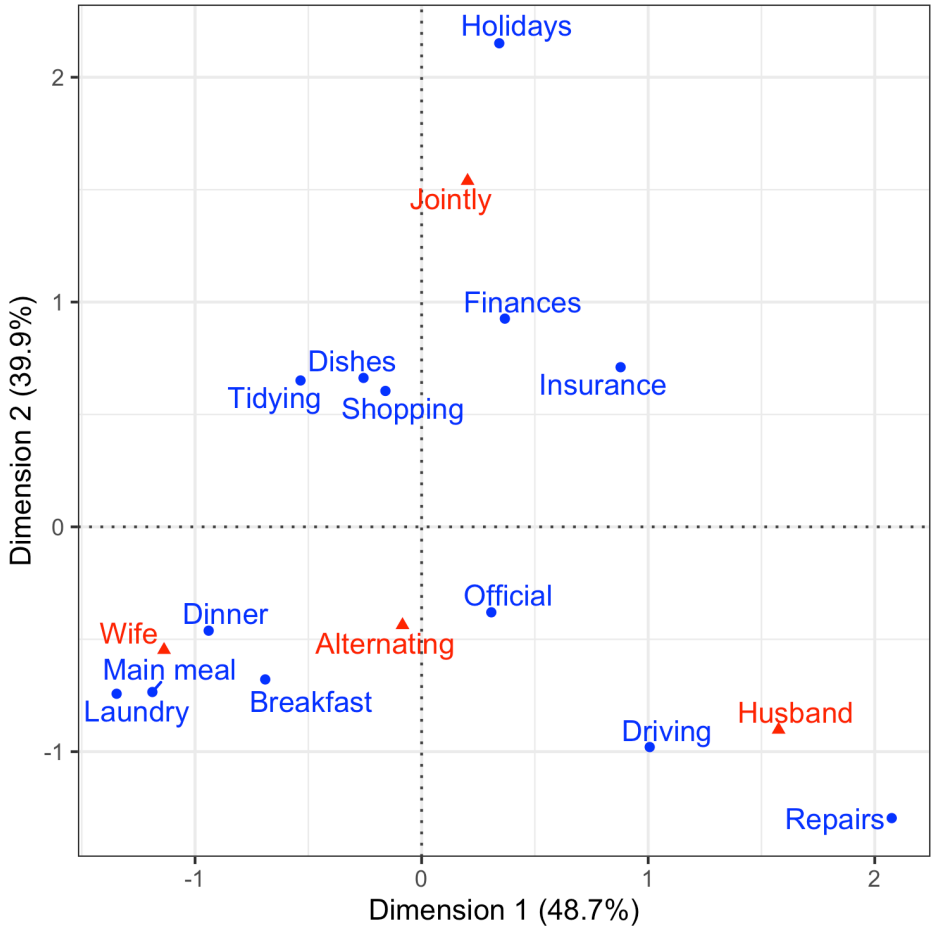

Figure 7-4. Graphical representation of a correspondence analysis of house task data

Key Ideas

- Principal components are linear combinations of the predictor variables (numeric data only).
- Principal components are calculated so as to minimize correlation between components, reducing redundancy.
- A limited number of components will typically explain most of the variance in the outcome variable.
- The limited set of principal components can then be used in place of the (more numerous) original predictors, reducing dimensionality.
- A superficially similar technique for categorical data is correspondence analysis, but it is not useful in a big data context.

Further Reading

For a detailed look at the use of cross-validation in principal components, see Rasmus Bro, K. Kjeldahl, A.K. Smilde, and Henk A. L. Kiers, "Cross-Validation of Component Models: A Critical Look at Current Methods", *Analytical and Bioanalytical Chemistry* 390, no. 5 (2008).

K-Means Clustering

Clustering is a technique to divide data into different groups, where the records in each group are similar to one another. A goal of clustering is to identify significant and meaningful groups of data. The groups can be used directly, analyzed in more depth, or passed as a feature or an outcome to a predictive regression or classification model. *K-means* was the first clustering method to be developed; it is still widely used, owing its popularity to the relative simplicity of the algorithm and its ability to scale to large data sets.

Key Terms for K-Means Clustering

Cluster
: A group of records that are similar.

Cluster mean
: The vector of variable means for the records in a cluster.

K
: The number of clusters.

K-means divides the data into *K* clusters by minimizing the sum of the squared distances of each record to the *mean* of its assigned cluster. This is referred to as the *within-cluster sum of squares* or *within-cluster SS*. *K*-means does not ensure the clusters will have the same size but finds the clusters that are the best separated.

Normalization

It is typical to normalize (standardize) continuous variables by subtracting the mean and dividing by the standard deviation. Otherwise, variables with large scale will dominate the clustering process (see "Standardization (Normalization, z-Scores)" on page 243).

A Simple Example

Start by considering a data set with n records and just two variables, x and y. Suppose we want to split the data into $K = 4$ clusters. This means assigning each record (x_i, y_i) to a cluster k. Given an assignment of n_k records to cluster k, the center of the cluster $(\bar{x}_k, \bar{y}_k)$ is the mean of the points in the cluster:

$$\bar{x}_k = \frac{1}{n_k} \sum_{i \in \text{Cluster } k} x_i$$

$$\bar{y}_k = \frac{1}{n_k} \sum_{i \in \text{Cluster } k} y_i$$

Cluster Mean

In clustering records with multiple variables (the typical case), the term *cluster mean* refers not to a single number but to the vector of means of the variables.

The sum of squares within a cluster is given by:

$$\text{SS}_k = \sum_{i \in \text{Cluster } k} (x_i - \bar{x}_k)^2 + (y_i - \bar{y}_k)^2$$

K-means finds the assignment of records that minimizes within-cluster sum of squares across all four clusters $\text{SS}_1 + \text{SS}_2 + \text{SS}_3 + \text{SS}_4$:

$$\sum_{k=1}^{4} \text{SS}_k$$

A typical use of clustering is to locate natural, separate clusters in the data. Another application is to divide the data into a predetermined number of separate groups, where clustering is used to ensure the groups are as different as possible from one another.

For example, suppose we want to divide daily stock returns into four groups. *K*-means clustering can be used to separate the data into the best groupings. Note that daily stock returns are reported in a fashion that is, in effect, standardized, so we do not need to normalize the data. In *R*, *K*-means clustering can be performed using the `kmeans` function. For example, the following finds four clusters based on two variables—the daily stock returns for ExxonMobil (`XOM`) and Chevron (`CVX`):

```
df <- sp500_px[row.names(sp500_px)>='2011-01-01', c('XOM', 'CVX')]
km <- kmeans(df, centers=4)
```

We use the `sklearn.cluster.KMeans` method from `scikit-learn` in *Python*:

```
df = sp500_px.loc[sp500_px.index >= '2011-01-01', ['XOM', 'CVX']]
kmeans = KMeans(n_clusters=4).fit(df)
```

The cluster assignment for each record is returned as the `cluster` component (*R*):

```
> df$cluster <- factor(km$cluster)
> head(df)
                 XOM        CVX cluster
2011-01-03 0.73680496  0.2406809       2
2011-01-04 0.16866845 -0.5845157       1
2011-01-05 0.02663055  0.4469854       2
2011-01-06 0.24855834 -0.9197513       1
2011-01-07 0.33732892  0.1805111       2
2011-01-10 0.00000000 -0.4641675       1
```

In `scikit-learn`, the cluster labels are available in the `labels_` field:

```
df['cluster'] = kmeans.labels_
df.head()
```

The first six records are assigned to either cluster 1 or cluster 2. The means of the clusters are also returned (*R*):

```
> centers <- data.frame(cluster=factor(1:4), km$centers)
> centers
  cluster        XOM        CVX
1       1 -0.3284864 -0.5669135
2       2  0.2410159  0.3342130
3       3 -1.1439800 -1.7502975
4       4  0.9568628  1.3708892
```

In `scikit-learn`, the cluster centers are available in the `cluster_centers_` field:

```
centers = pd.DataFrame(kmeans.cluster_centers_, columns=['XOM', 'CVX'])
centers
```

Clusters 1 and 3 represent "down" markets, while clusters 2 and 4 represent "up markets."

As the *K*-means algorithm uses randomized starting points, the results may differ between subsequent runs and different implementations of the method. In general, you should check that the fluctuations aren't too large.

In this example, with just two variables, it is straightforward to visualize the clusters and their means:

```
ggplot(data=df, aes(x=XOM, y=CVX, color=cluster, shape=cluster)) +
  geom_point(alpha=.3) +
  geom_point(data=centers,  aes(x=XOM, y=CVX), size=3, stroke=2)
```

The `seaborn` `scatterplot` function makes it easy to color (`hue`) and style (`style`) the points by a property:

```
fig, ax = plt.subplots(figsize=(4, 4))
ax = sns.scatterplot(x='XOM', y='CVX', hue='cluster', style='cluster',
                     ax=ax, data=df)
ax.set_xlim(-3, 3)
ax.set_ylim(-3, 3)
centers.plot.scatter(x='XOM', y='CVX', ax=ax, s=50, color='black')
```

The resulting plot, shown in Figure 7-5, shows the cluster assignments and the cluster means. Note that *K*-means will assign records to clusters, even if those clusters are not well separated (which can be useful if you need to optimally divide records into groups).

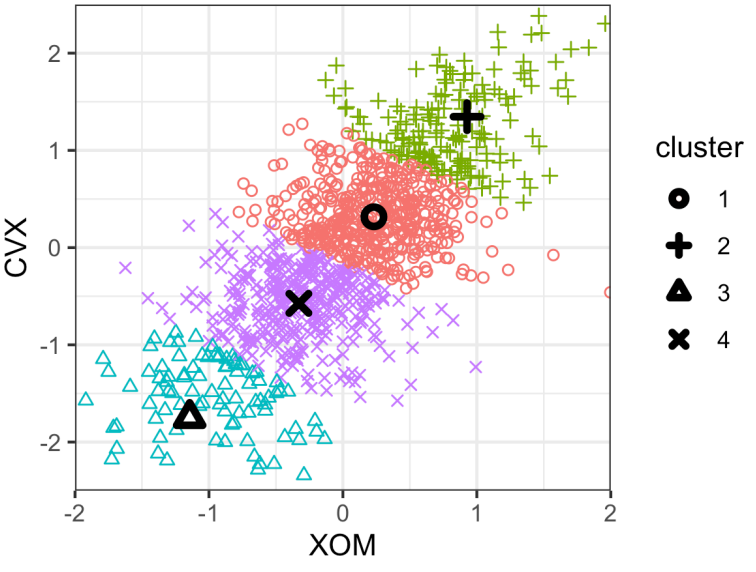

Figure 7-5. The clusters of K-means applied to daily stock returns for ExxonMobil and Chevron (the cluster centers are highlighted with black symbols)

K-Means Algorithm

In general, *K*-means can be applied to a data set with p variables $X_1, ..., X_p$. While the exact solution to *K*-means is computationally very difficult, heuristic algorithms provide an efficient way to compute a locally optimal solution.

The algorithm starts with a user-specified K and an initial set of cluster means and then iterates the following steps:

1. Assign each record to the nearest cluster mean as measured by squared distance.
2. Compute the new cluster means based on the assignment of records.

The algorithm converges when the assignment of records to clusters does not change.

For the first iteration, you need to specify an initial set of cluster means. Usually you do this by randomly assigning each record to one of the K clusters and then finding the means of those clusters.

Since this algorithm isn't guaranteed to find the best possible solution, it is recommended to run the algorithm several times using different random samples to initialize the algorithm. When more than one set of iterations is used, the *K*-means result is given by the iteration that has the lowest within-cluster sum of squares.

The `nstart` parameter to the *R* function `kmeans` allows you to specify the number of random starts to try. For example, the following code runs *K*-means to find 5 clusters using 10 different starting cluster means:

```
syms <- c( 'AAPL', 'MSFT', 'CSCO', 'INTC', 'CVX', 'XOM', 'SLB', 'COP',
           'JPM', 'WFC', 'USB', 'AXP', 'WMT', 'TGT', 'HD', 'COST')
df <- sp500_px[row.names(sp500_px) >= '2011-01-01', syms]
km <- kmeans(df, centers=5, nstart=10)
```

The function automatically returns the best solution out of the 10 different starting points. You can use the argument `iter.max` to set the maximum number of iterations the algorithm is allowed for each random start.

The `scikit-learn` algorithm is repeated 10 times by default (`n_init`). The argument `max_iter` (default 300) can be used to control the number of iterations:

```
syms = sorted(['AAPL', 'MSFT', 'CSCO', 'INTC', 'CVX', 'XOM', 'SLB', 'COP',
               'JPM', 'WFC', 'USB', 'AXP', 'WMT', 'TGT', 'HD', 'COST'])
top_sp = sp500_px.loc[sp500_px.index >= '2011-01-01', syms]
kmeans = KMeans(n_clusters=5).fit(top_sp)
```

Interpreting the Clusters

An important part of cluster analysis can involve the interpretation of the clusters. The two most important outputs from `kmeans` are the sizes of the clusters and the cluster means. For the example in the previous subsection, the sizes of resulting clusters are given by this *R* command:

```
km$size
[1] 106 186 285 288 266
```

In *Python*, we can use the `collections.Counter` class from the standard library to get this information. Due to differences in the implementation and the inherent randomness of the algorithm, results will vary:

```
from collections import Counter
Counter(kmeans.labels_)

Counter({4: 302, 2: 272, 0: 288, 3: 158, 1: 111})
```

The cluster sizes are relatively balanced. Imbalanced clusters can result from distant outliers, or from groups of records very distinct from the rest of the data—both may warrant further inspection.

You can plot the centers of the clusters using the `gather` function in conjunction with `ggplot`:

```
centers <- as.data.frame(t(centers))
names(centers) <- paste("Cluster", 1:5)
centers$Symbol <- row.names(centers)
centers <- gather(centers, 'Cluster', 'Mean', -Symbol)
centers$Color = centers$Mean > 0
ggplot(centers, aes(x=Symbol, y=Mean, fill=Color)) +
  geom_bar(stat='identity', position='identity', width=.75) +
  facet_grid(Cluster ~ ., scales='free_y')
```

The code to create this visualization in *Python* is similar to what we used for PCA:

```
centers = pd.DataFrame(kmeans.cluster_centers_, columns=syms)

f, axes = plt.subplots(5, 1, figsize=(5, 5), sharex=True)
for i, ax in enumerate(axes):
    center = centers.loc[i, :]
    maxPC = 1.01 * np.max(np.max(np.abs(center)))
    colors = ['C0' if l > 0 else 'C1' for l in center]
    ax.axhline(color='#888888')
    center.plot.bar(ax=ax, color=colors)
    ax.set_ylabel(f'Cluster {i + 1}')
    ax.set_ylim(-maxPC, maxPC)
```

The resulting plot is shown in Figure 7-6 and reveals the nature of each cluster. For example, clusters 4 and 5 correspond to days on which the market is down and up, respectively. Clusters 2 and 3 are characterized by up-market days for consumer stocks and down-market days for energy stocks, respectively. Finally, cluster 1 captures the days in which energy stocks were up and consumer stocks were down.

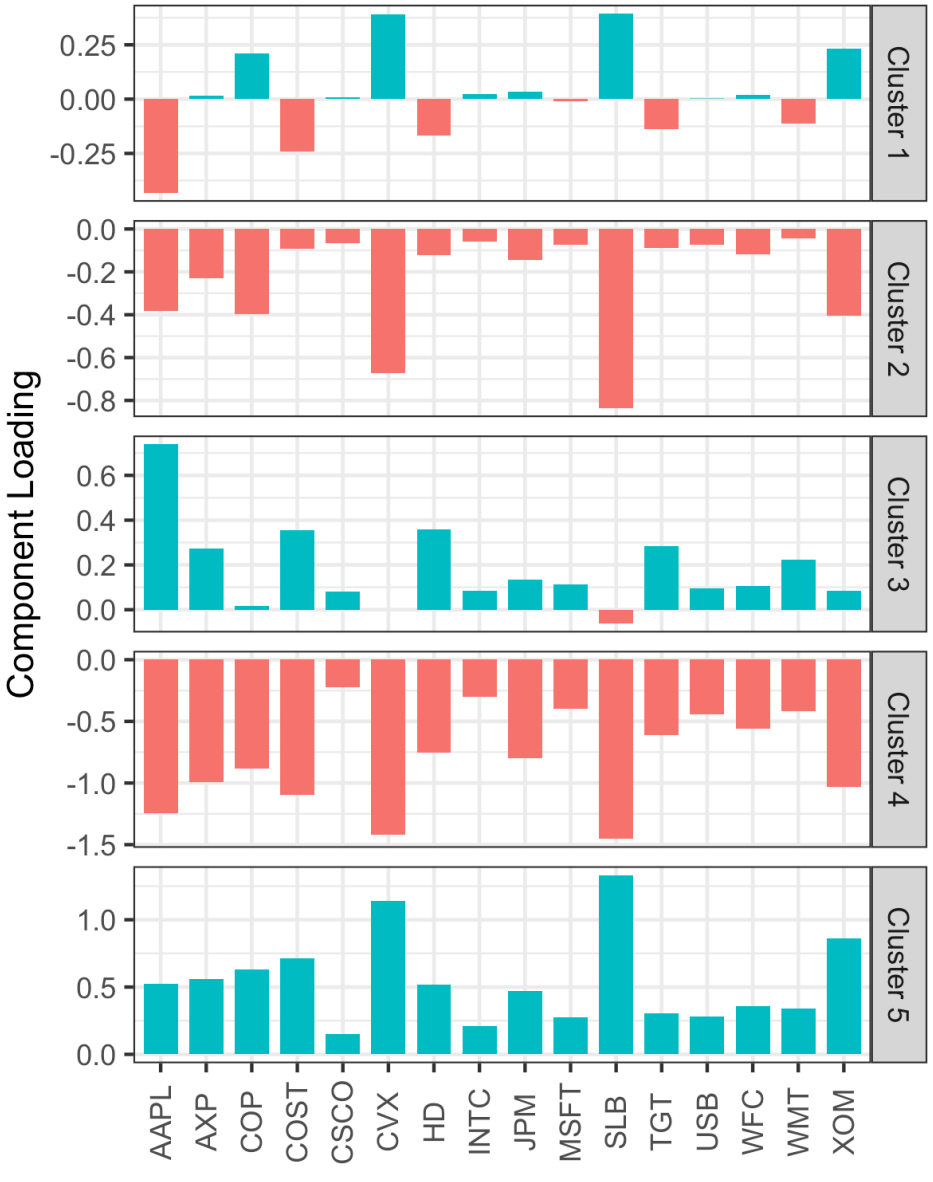

Figure 7-6. The means of the variables in each cluster ("cluster means")

Cluster Analysis Versus PCA

The plot of cluster means is similar in spirit to looking at the loadings for principal components analysis (PCA); see "Interpreting Principal Components" on page 289. A major distinction is that unlike with PCA, the sign of the cluster means is meaningful. PCA identifies principal directions of variation, whereas cluster analysis finds groups of records located near one another.

Selecting the Number of Clusters

The *K*-means algorithm requires that you specify the number of clusters *K*. Sometimes the number of clusters is driven by the application. For example, a company managing a sales force might want to cluster customers into "personas" to focus and guide sales calls. In such a case, managerial considerations would dictate the number of desired customer segments—for example, two might not yield useful differentiation of customers, while eight might be too many to manage.

In the absence of a cluster number dictated by practical or managerial considerations, a statistical approach could be used. There is no single standard method to find the "best" number of clusters.

A common approach, called the *elbow method*, is to identify when the set of clusters explains "most" of the variance in the data. Adding new clusters beyond this set contributes relatively little in the variance explained. The elbow is the point where the cumulative variance explained flattens out after rising steeply, hence the name of the method.

Figure 7-7 shows the cumulative percent of variance explained for the default data for the number of clusters ranging from 2 to 15. Where is the elbow in this example? There is no obvious candidate, since the incremental increase in variance explained drops gradually. This is fairly typical in data that does not have well-defined clusters. This is perhaps a drawback of the elbow method, but it does reveal the nature of the data.

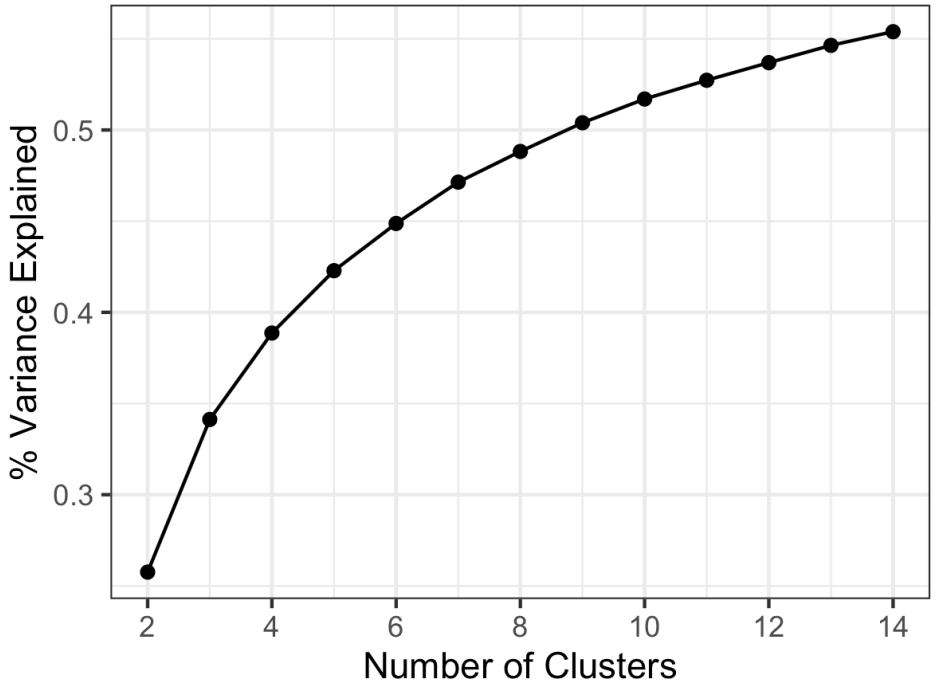

Figure 7-7. The elbow method applied to the stock data

In *R*, the `kmeans` function doesn't provide a single command for applying the elbow method, but it can be readily applied from the output of `kmeans` as shown here:

```
pct_var <- data.frame(pct_var = 0,
                      num_clusters = 2:14)
totalss <- kmeans(df, centers=14, nstart=50, iter.max=100)$totss
for (i in 2:14) {
  kmCluster <- kmeans(df, centers=i, nstart=50, iter.max=100)
  pct_var[i-1, 'pct_var'] <- kmCluster$betweenss / totalss
}
```

For the `KMeans` result, we get this information from the property `inertia_`. After conversion into a `pandas` data frame, we can use its `plot` method to create the graph:

```
inertia = []
for n_clusters in range(2, 14):
    kmeans = KMeans(n_clusters=n_clusters, random_state=0).fit(top_sp)
    inertia.append(kmeans.inertia_ / n_clusters)

inertias = pd.DataFrame({'n_clusters': range(2, 14), 'inertia': inertia})
ax = inertias.plot(x='n_clusters', y='inertia')
plt.xlabel('Number of clusters(k)')
plt.ylabel('Average Within-Cluster Squared Distances')
```

```
plt.ylim((0, 1.1 * inertias.inertia.max()))
ax.legend().set_visible(False)
```

In evaluating how many clusters to retain, perhaps the most important test is this: how likely are the clusters to be replicated on new data? Are the clusters interpretable, and do they relate to a general characteristic of the data, or do they just reflect a specific instance? You can assess this, in part, using cross-validation; see "Cross-Validation" on page 155.

In general, there is no single rule that will reliably guide how many clusters to produce.

There are several more formal ways to determine the number of clusters based on statistical or information theory. For example, Robert Tibshirani, Guenther Walther, and Trevor Hastie propose a "gap" statistic based on statistical theory to identify the elbow. For most applications, a theoretical approach is probably not necessary, or even appropriate.

Key Ideas

- The number of desired clusters, *K*, is chosen by the user.
- The algorithm develops clusters by iteratively assigning records to the nearest cluster mean until cluster assignments do not change.
- Practical considerations usually dominate the choice of *K*; there is no statistically determined optimal number of clusters.

Hierarchical Clustering

Hierarchical clustering is an alternative to *K*-means that can yield very different clusters. Hierarchical clustering allows the user to visualize the effect of specifying different numbers of clusters. It is more sensitive in discovering outlying or aberrant groups or records. Hierarchical clustering also lends itself to an intuitive graphical display, leading to easier interpretation of the clusters.

Key Terms for Hierarchical Clustering

Dendrogram
A visual representation of the records and the hierarchy of clusters to which they belong.

Distance
A measure of how close one *record* is to another.

Dissimilarity
A measure of how close one *cluster* is to another.

Hierarchical clustering's flexibility comes with a cost, and hierarchical clustering does not scale well to large data sets with millions of records. For even modest-sized data with just tens of thousands of records, hierarchical clustering can require intensive computing resources. Indeed, most of the applications of hierarchical clustering are focused on relatively small data sets.

A Simple Example

Hierarchical clustering works on a data set with n records and p variables and is based on two basic building blocks:

- A distance metric $d_{i,j}$ to measure the distance between two records i and j.
- A dissimilarity metric $D_{A,B}$ to measure the difference between two clusters A and B based on the distances $d_{i,j}$ between the members of each cluster.

For applications involving numeric data, the most importance choice is the dissimilarity metric. Hierarchical clustering starts by setting each record as its own cluster and iterates to combine the least dissimilar clusters.

In *R*, the `hclust` function can be used to perform hierarchical clustering. One big difference with `hclust` versus `kmeans` is that it operates on the pairwise distances $d_{i,j}$ rather than the data itself. You can compute these using the `dist` function. For example, the following applies hierarchical clustering to the stock returns for a set of companies:

```
syms1 <- c('GOOGL', 'AMZN', 'AAPL', 'MSFT', 'CSCO', 'INTC', 'CVX', 'XOM', 'SLB',
           'COP', 'JPM', 'WFC', 'USB', 'AXP', 'WMT', 'TGT', 'HD', 'COST')
# take transpose: to cluster companies, we need the stocks along the rows
df <- t(sp500_px[row.names(sp500_px) >= '2011-01-01', syms1])
d <- dist(df)
hcl <- hclust(d)
```

Clustering algorithms will cluster the records (rows) of a data frame. Since we want to cluster the companies, we need to *transpose* (`t`) the data frame and put the stocks along the rows and the dates along the columns.

The `scipy` package offers a number of different methods for hierarchical clustering in the `scipy.cluster.hierarchy` module. Here we use the `linkage` function with the "complete" method:

```
syms1 = ['AAPL', 'AMZN', 'AXP', 'COP', 'COST', 'CSCO', 'CVX', 'GOOGL', 'HD',
         'INTC', 'JPM', 'MSFT', 'SLB', 'TGT', 'USB', 'WFC', 'WMT', 'XOM']
df = sp500_px.loc[sp500_px.index >= '2011-01-01', syms1].transpose()

Z = linkage(df, method='complete')
```

The Dendrogram

Hierarchical clustering lends itself to a natural graphical display as a tree, referred to as a *dendrogram*. The name comes from the Greek words *dendro* (tree) and *gramma* (drawing). In *R*, you can easily produce this using the `plot` command:

```
plot(hcl)
```

We can use the `dendrogram` method to plot the result of the `linkage` function in *Python*:

```
fig, ax = plt.subplots(figsize=(5, 5))
dendrogram(Z, labels=df.index, ax=ax, color_threshold=0)
plt.xticks(rotation=90)
ax.set_ylabel('distance')
```

The result is shown in Figure 7-8 (note that we are now plotting companies that are similar to one another, not days). The leaves of the tree correspond to the records. The length of the branch in the tree indicates the degree of dissimilarity between corresponding clusters. The returns for Google and Amazon are quite dissimilar to one another and to the returns for the other stocks. The oil stocks (SLB, CVX, XOM, COP) are in their own cluster, Apple (AAPL) is by itself, and the rest are similar to one another.

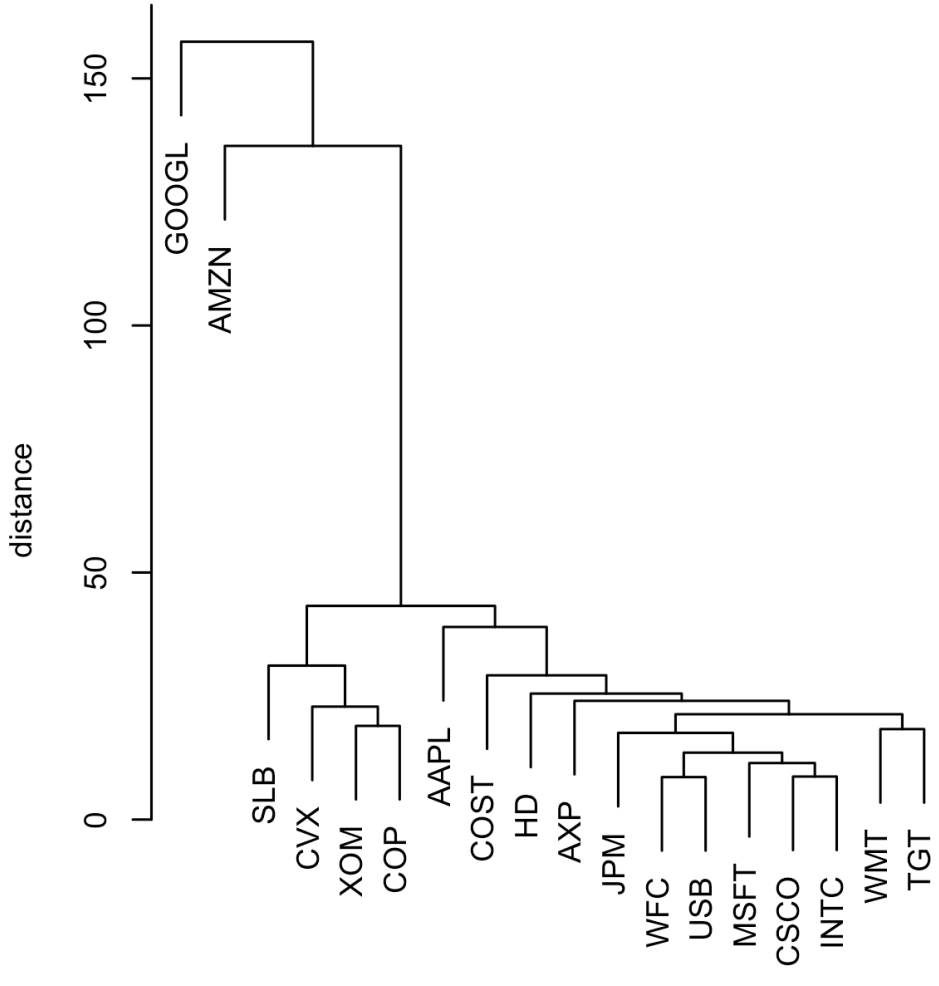

Figure 7-8. A dendrogram of stocks

In contrast to *K*-means, it is not necessary to prespecify the number of clusters. Graphically, you can identify different numbers of clusters with a horizontal line that slides up or down; a cluster is defined wherever the horizontal line intersects the vertical lines. To extract a specific number of clusters, you can use the `cutree` function:

```
cutree(hcl, k=4)
GOOGL  AMZN  AAPL  MSFT  CSCO  INTC   CVX   XOM   SLB   COP   JPM   WFC
    1     2     3     3     3     3     4     4     4     4     3     3
  USB   AXP   WMT   TGT    HD  COST
    3     3     3     3     3     3
```

In *Python*, you achieve the same with the `fcluster` method:

```
memb = fcluster(Z, 4, criterion='maxclust')
memb = pd.Series(memb, index=df.index)
for key, item in memb.groupby(memb):
    print(f"{key} : {', '.join(item.index)}")
```

The number of clusters to extract is set to 4, and you can see that Google and Amazon each belong to their own cluster. The oil stocks all belong to another cluster. The remaining stocks are in the fourth cluster.

The Agglomerative Algorithm

The main algorithm for hierarchical clustering is the *agglomerative* algorithm, which iteratively merges similar clusters. The agglomerative algorithm begins with each record constituting its own single-record cluster and then builds up larger and larger clusters. The first step is to calculate distances between all pairs of records.

For each pair of records $\left(x_1, x_2, ..., x_p\right)$ and $\left(y_1, y_2, ..., y_p\right)$, we measure the distance between the two records, $d_{x,y}$, using a distance metric (see "Distance Metrics" on page 241). For example, we can use Euclidian distance:

$$d(x,y) = \sqrt{\left(x_1 - y_1\right)^2 + \left(x_2 - y_2\right)^2 + \cdots + \left(x_p - y_p\right)^2}$$

We now turn to inter-cluster distance. Consider two clusters A and B, each with a distinctive set of records, $A = \left(a_1, a_2, ..., a_m\right)$ and $B = \left(b_1, b_2, ..., b_q\right)$. We can measure the dissimilarity between the clusters $D(A, B)$ by using the distances between the members of A and the members of B.

One measure of dissimilarity is the *complete-linkage* method, which is the maximum distance across all pairs of records between A and B:

$$D(A,B) = \max d\left(a_i, b_j\right) \text{ for all pairs } i, j$$

This defines the dissimilarity as the biggest difference between all pairs.

The main steps of the agglomerative algorithm are:

1. Create an initial set of clusters with each cluster consisting of a single record for all records in the data.
2. Compute the dissimilarity $D(C_k, C_\ell)$ between all pairs of clusters k, ℓ.
3. Merge the two clusters C_k and C_ℓ that are least dissimilar as measured by $D(C_k, C_\ell)$.
4. If we have more than one cluster remaining, return to step 2. Otherwise, we are done.

Measures of Dissimilarity

There are four common measures of dissimilarity: *complete linkage*, *single linkage*, *average linkage*, and *minimum variance*. These (plus other measures) are all supported by most hierarchical clustering software, including `hclust` and `linkage`. The complete linkage method defined earlier tends to produce clusters with members that are similar. The single linkage method is the minimum distance between the records in two clusters:

$$D(A, B) = \min d\left(a_i, b_j\right) \text{ for all pairs } i, j$$

This is a "greedy" method and produces clusters that can contain quite disparate elements. The average linkage method is the average of all distance pairs and represents a compromise between the single and complete linkage methods. Finally, the minimum variance method, also referred to as *Ward's* method, is similar to *K*-means since it minimizes the within-cluster sum of squares (see "K-Means Clustering" on page 294).

Figure 7-9 applies hierarchical clustering using the four measures to the ExxonMobil and Chevron stock returns. For each measure, four clusters are retained.

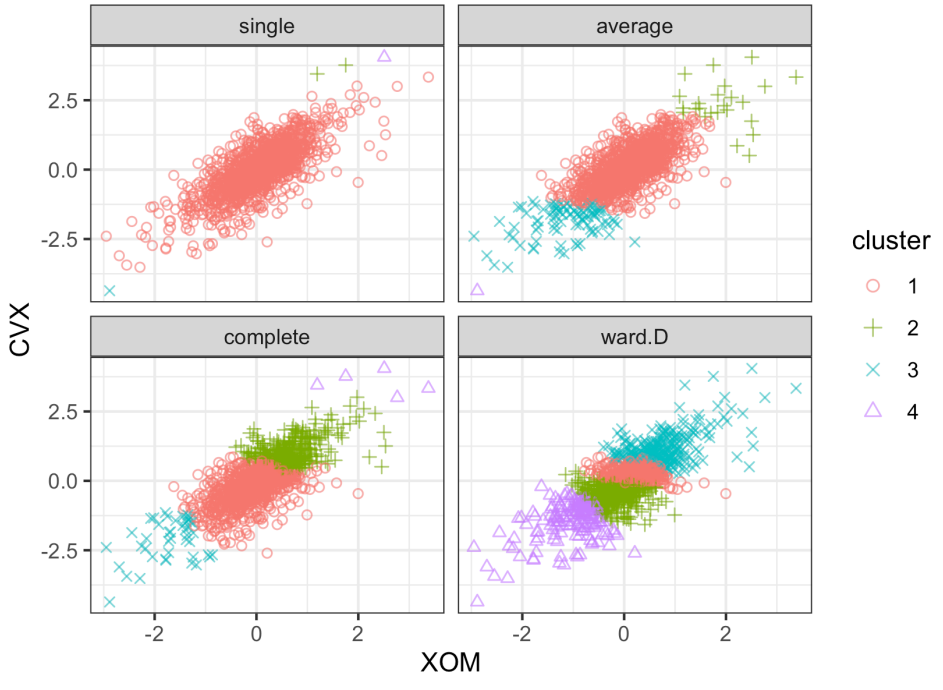

Figure 7-9. A comparison of measures of dissimilarity applied to stock data

The results are strikingly different: the single linkage measure assigns almost all of the points to a single cluster. Except for the minimum variance method (*R*: `Ward.D`; *Python*: `ward`), all measures end up with at least one cluster with just a few outlying points. The minimum variance method is most similar to the *K*-means cluster; compare with Figure 7-5.

Key Ideas

- Hierarchical clustering starts with every record in its own cluster.
- Progressively, clusters are joined to nearby clusters until all records belong to a single cluster (the agglomerative algorithm).
- The agglomeration history is retained and plotted, and the user (without specifying the number of clusters beforehand) can visualize the number and structure of clusters at different stages.
- Inter-cluster distances are computed in different ways, all relying on the set of all inter-record distances.

Model-Based Clustering

Clustering methods such as hierarchical clustering and *K*-means are based on heuristics and rely primarily on finding clusters whose members are close to one another, as measured directly with the data (no probability model involved). In the past 20 years, significant effort has been devoted to developing *model-based clustering* methods. Adrian Raftery and other researchers at the University of Washington made critical contributions to model-based clustering, including both theory and software. The techniques are grounded in statistical theory and provide more rigorous ways to determine the nature and number of clusters. They could be used, for example, in cases where there might be one group of records that are similar to one another but not necessarily close to one another (e.g., tech stocks with high variance of returns), and another group of records that are similar and also close (e.g., utility stocks with low variance).

Multivariate Normal Distribution

The most widely used model-based clustering methods rest on the *multivariate normal* distribution. The multivariate normal distribution is a generalization of the normal distribution to a set of p variables $X_1, X_2, ..., X_p$. The distribution is defined by a set of means $\mu = \mu_\mathbf{1}, \mu_\mathbf{2}, ..., \mu_\mathbf{p}$ and a covariance matrix Σ. The covariance matrix is a measure of how the variables correlate with each other (see "Covariance Matrix" on page 202 for details on the covariance). The covariance matrix Σ consists of p variances $\sigma_1^2, \sigma_2^2, ..., \sigma_p^2$ and covariances $\sigma_{i,j}$ for all pairs of variables $i \neq j$. With the variables put along the rows and duplicated along the columns, the matrix looks like this:

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{1,2} & \cdots & \sigma_{1,p} \\ \sigma_{2,1} & \sigma_2^2 & \cdots & \sigma_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{p,1} & \sigma_{p,2}^2 & \cdots & \sigma_p^2 \end{bmatrix}$$

Note that the covariance matrix is symmetric around the diagonal from upper left to lower right. Since $\sigma_{i,j} = \sigma_{j,i}$, there are only $(p \times (p-1))/2$ covariance terms. In total, the covariance matrix has $(p \times (p-1))/2 + p$ parameters. The distribution is denoted by:

$$\left(X_1, X_2, ..., X_p\right) \sim N_p(\mu, \Sigma)$$

This is a symbolic way of saying that the variables are all normally distributed, and the overall distribution is fully described by the vector of variable means and the covariance matrix.

Figure 7-10 shows the probability contours for a multivariate normal distribution for two variables X and Y (the 0.5 probability contour, for example, contains 50% of the distribution).

The means are $\mu_x = 0.5$ and $\mu_y = -0.5$, and the covariance matrix is:

$$\Sigma = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$$

Since the covariance σ_{xy} is positive, X and Y are positively correlated.

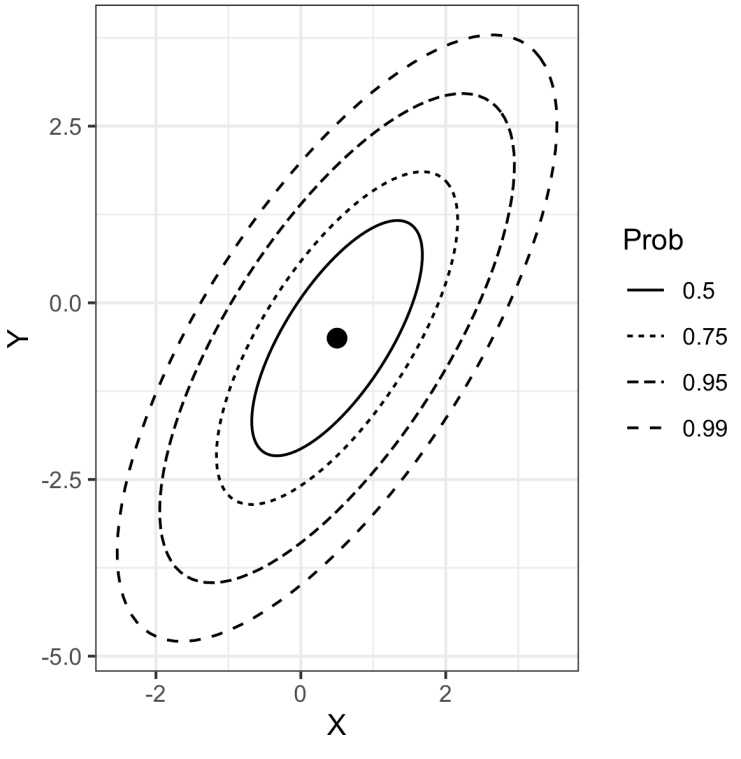

Figure 7-10. Probability contours for a two-dimensional normal distribution

Mixtures of Normals

The key idea behind model-based clustering is that each record is assumed to be distributed as one of K multivariate normal distributions, where K is the number of

clusters. Each distribution has a different mean μ and covariance matrix Σ. For example, if you have two variables, X and Y, then each row (X_i, Y_i) is modeled as having been sampled from one of K multivariate normal distributions $N(\mu_1, \Sigma_1), N(\mu_2, \Sigma_2), ..., N(\mu_K, \Sigma_K)$.

R has a very rich package for model-based clustering called `mclust`, originally developed by Chris Fraley and Adrian Raftery. With this package, we can apply model-based clustering to the stock return data we previously analyzed using *K*-means and hierarchical clustering:

```
> library(mclust)
> df <- sp500_px[row.names(sp500_px) >= '2011-01-01', c('XOM', 'CVX')]
> mcl <- Mclust(df)
> summary(mcl)
Mclust VEE (ellipsoidal, equal shape and orientation) model with 2 components:


 log.likelihood    n df       BIC       ICL
      -2255.134 1131  9 -4573.546 -5076.856

Clustering table:
  1   2
963 168
```

`scikit-learn` has the `sklearn.mixture.GaussianMixture` class for model-based clustering:

```
df = sp500_px.loc[sp500_px.index >= '2011-01-01', ['XOM', 'CVX']]
mclust = GaussianMixture(n_components=2).fit(df)
mclust.bic(df)
```

If you execute this code, you will notice that the computation takes significantly longer than other procedures. Extracting the cluster assignments using the `predict` function, we can visualize the clusters:

```
cluster <- factor(predict(mcl)$classification)
ggplot(data=df, aes(x=XOM, y=CVX, color=cluster, shape=cluster)) +
  geom_point(alpha=.8)
```

Here is the *Python* code to create a similar figure:

```
fig, ax = plt.subplots(figsize=(4, 4))
colors = [f'C{c}' for c in mclust.predict(df)]
df.plot.scatter(x='XOM', y='CVX', c=colors, alpha=0.5, ax=ax)
ax.set_xlim(-3, 3)
ax.set_ylim(-3, 3)
```

The resulting plot is shown in Figure 7-11. There are two clusters: one cluster in the middle of the data, and a second cluster in the outer edge of the data. This is very different from the clusters obtained using *K*-means (Figure 7-5) and hierarchical clustering (Figure 7-9), which find clusters that are compact.

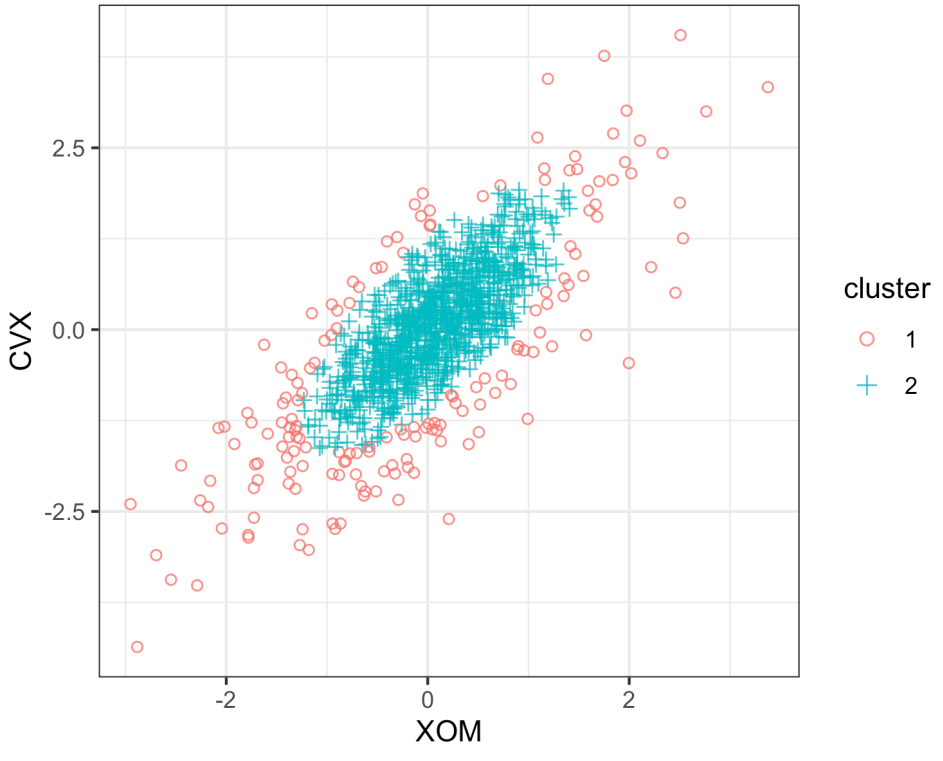

Figure 7-11. Two clusters are obtained for stock return data using `mclust`

You can extract the parameters to the normal distributions using the `summary` function:

```
> summary(mcl, parameters=TRUE)$mean
          [,1]        [,2]
XOM 0.05783847 -0.04374944
CVX 0.07363239 -0.21175715
> summary(mcl, parameters=TRUE)$variance
, , 1
          XOM       CVX
XOM 0.3002049 0.3060989
CVX 0.3060989 0.5496727
, , 2

         XOM      CVX
XOM 1.046318 1.066860
CVX 1.066860 1.915799
```

In *Python*, you get this information from the `means_` and `covariances_` properties of the result:

```
print('Mean')
print(mclust.means_)
print('Covariances')
print(mclust.covariances_)
```

The distributions have similar means and correlations, but the second distribution has much larger variances and covariances. Due to the randomness of the algorithm, results can vary slightly between different runs.

The clusters from `mclust` may seem surprising, but in fact, they illustrate the statistical nature of the method. The goal of model-based clustering is to find the best-fitting set of multivariate normal distributions. The stock data appears to have a normal-looking shape: see the contours of Figure 7-10. In fact, though, stock returns have a longer-tailed distribution than a normal distribution. To handle this, `mclust` fits a distribution to the bulk of the data but then fits a second distribution with a bigger variance.

Selecting the Number of Clusters

Unlike *K*-means and hierarchical clustering, `mclust` automatically selects the number of clusters in *R* (in this case, two). It does this by choosing the number of clusters for which the *Bayesian Information Criteria* (*BIC*) has the largest value (BIC is similar to AIC; see "Model Selection and Stepwise Regression" on page 156). BIC works by selecting the best-fitting model with a penalty for the number of parameters in the model. In the case of model-based clustering, adding more clusters will always improve the fit at the expense of introducing additional parameters in the model.

Note that in most cases BIC is usually minimized. The authors of the `mclust` package decided to define BIC to have the opposite sign to make interpretation of plots easier.

`mclust` fits 14 different models with increasing number of components and chooses an optimal model automatically. You can plot the BIC values of these models using a function in `mclust`:

```
plot(mcl, what='BIC', ask=FALSE)
```

The number of clusters—or number of different multivariate normal models (components)—is shown on the x-axis (see Figure 7-12).

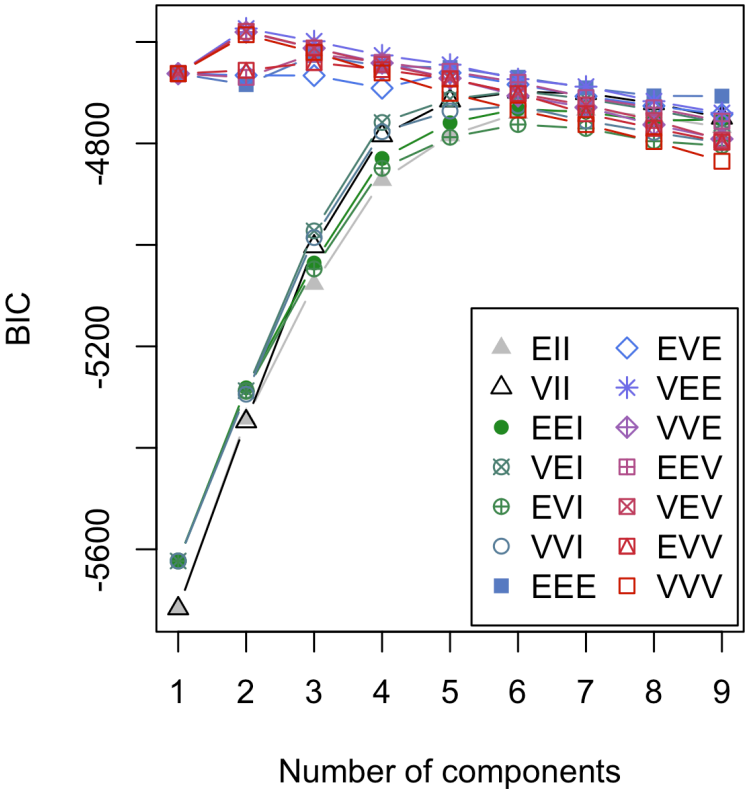

Figure 7-12. BIC values for 14 models of the stock return data with increasing numbers of components

The `GaussianMixture` implementation on the other hand will not try out various combinations. As shown here, it is straightforward to run multiple combinations using *Python*. This implementation defines BIC as usual. Therefore, the calculated BIC value will be positive, and we need to minimize it.

```
results = []
covariance_types = ['full', 'tied', 'diag', 'spherical']
for n_components in range(1, 9):
    for covariance_type in covariance_types:
        mclust = GaussianMixture(n_components=n_components, warm_start=True,
                                 covariance_type=covariance_type) ❶
        mclust.fit(df)
        results.append({
            'bic': mclust.bic(df),
            'n_components': n_components,
            'covariance_type': covariance_type,
        })
```

```
results = pd.DataFrame(results)

colors = ['C0', 'C1', 'C2', 'C3']
styles = ['C0-','C1:','C0-.', 'C1--']

fig, ax = plt.subplots(figsize=(4, 4))
for i, covariance_type in enumerate(covariance_types):
    subset = results.loc[results.covariance_type == covariance_type, :]
    subset.plot(x='n_components', y='bic', ax=ax, label=covariance_type,
                kind='line', style=styles[i])
```

❶ With the `warm_start` argument, the calculation will reuse information from the previous fit. This will speed up the convergence of subsequent calculations.

This plot is similar to the elbow plot used to identify the number of clusters to choose for *K*-means, except the value being plotted is BIC instead of percent of variance explained (see Figure 7-7). One big difference is that instead of one line, `mclust` shows 14 different lines! This is because `mclust` is actually fitting 14 different models for each cluster size, and ultimately it chooses the best-fitting model. `GaussianMixture` implements fewer approaches, so the number of lines will be only four.

Why does `mclust` fit so many models to determine the best set of multivariate normals? It's because there are different ways to parameterize the covariance matrix Σ for fitting a model. For the most part, you do not need to worry about the details of the models and can simply use the model chosen by `mclust`. In this example, according to BIC, three different models (called VEE, VEV, and VVE) give the best fit using two components.

Model-based clustering is a rich and rapidly developing area of study, and the coverage in this text spans only a small part of the field. Indeed, the `mclust` help file is currently 154 pages long. Navigating the nuances of model-based clustering is probably more effort than is needed for most problems encountered by data scientists.

Model-based clustering techniques do have some limitations. The methods require an underlying assumption of a model for the data, and the cluster results are very dependent on that assumption. The computations requirements are higher than even hierarchical clustering, making it difficult to scale to large data. Finally, the algorithm is more sophisticated and less accessible than that of other methods.

Key Ideas

- Clusters are assumed to derive from different data-generating processes with different probability distributions.
- Different models are fit, assuming different numbers of (typically normal) distributions.
- The method chooses the model (and the associated number of clusters) that fits the data well without using too many parameters (i.e., overfitting).

Further Reading

For more detail on model-based clustering, see the `mclust` and `GaussianMixture` documentation.

Scaling and Categorical Variables

Unsupervised learning techniques generally require that the data be appropriately scaled. This is different from many of the techniques for regression and classification in which scaling is not important (an exception is *K*-Nearest Neighbors; see "K-Nearest Neighbors" on page 238).

Key Terms for Scaling Data

Scaling
: Squashing or expanding data, usually to bring multiple variables to the same scale.

Normalization
: One method of scaling—subtracting the mean and dividing by the standard deviation.

Synonym
: Standardization

Gower's distance
: A scaling algorithm applied to mixed numeric and categorical data to bring all variables to a 0–1 range.

For example, with the personal loan data, the variables have widely different units and magnitude. Some variables have relatively small values (e.g., number of years employed), while others have very large values (e.g., loan amount in dollars). If the

data is not scaled, then the PCA, *K*-means, and other clustering methods will be dominated by the variables with large values and ignore the variables with small values.

Categorical data can pose a special problem for some clustering procedures. As with *K*-Nearest Neighbors, unordered factor variables are generally converted to a set of binary (0/1) variables using one hot encoding (see "One Hot Encoder" on page 242). Not only are the binary variables likely on a different scale from other data, but the fact that binary variables have only two values can prove problematic with techniques such as PCA and *K*-means.

Scaling the Variables

Variables with very different scale and units need to be normalized appropriately before you apply a clustering procedure. For example, let's apply `kmeans` to a set of data of loan defaults without normalizing:

```
defaults <- loan_data[loan_data$outcome=='default',]
df <- defaults[, c('loan_amnt', 'annual_inc', 'revol_bal', 'open_acc',
                   'dti', 'revol_util')]
km <- kmeans(df, centers=4, nstart=10)
centers <- data.frame(size=km$size, km$centers)
round(centers, digits=2)
```

```
   size loan_amnt annual_inc revol_bal open_acc   dti revol_util
1    52  22570.19  489783.40  85161.35    13.33  6.91      59.65
2  1192  21856.38  165473.54  38935.88    12.61 13.48      63.67
3 13902  10606.48   42500.30  10280.52     9.59 17.71      58.11
4  7525  18282.25   83458.11  19653.82    11.66 16.77      62.27
```

Here is the corresponding *Python* code:

```
defaults = loan_data.loc[loan_data['outcome'] == 'default',]
columns = ['loan_amnt', 'annual_inc', 'revol_bal', 'open_acc',
           'dti', 'revol_util']

df = defaults[columns]
kmeans = KMeans(n_clusters=4, random_state=1).fit(df)
counts = Counter(kmeans.labels_)

centers = pd.DataFrame(kmeans.cluster_centers_, columns=columns)
centers['size'] = [counts[i] for i in range(4)]
centers
```

The variables `annual_inc` and `revol_bal` dominate the clusters, and the clusters have very different sizes. Cluster 1 has only 52 members with comparatively high income and revolving credit balance.

A common approach to scaling the variables is to convert them to *z*-scores by subtracting the mean and dividing by the standard deviation. This is termed

standardization or *normalization* (see "Standardization (Normalization, z-Scores)" on page 243 for more discussion about using *z*-scores):

$$z = \frac{x - \bar{x}}{s}$$

See what happens to the clusters when `kmeans` is applied to the normalized data:

```
df0 <- scale(df)
km0 <- kmeans(df0, centers=4, nstart=10)
centers0 <- scale(km0$centers, center=FALSE,
                  scale=1 / attr(df0, 'scaled:scale'))
centers0 <- scale(centers0, center=-attr(df0, 'scaled:center'), scale=FALSE)
centers0 <- data.frame(size=km0$size, centers0)
round(centers0, digits=2)
```

```
  size loan_amnt annual_inc revol_bal open_acc   dti revol_util
1 7355  10467.65   51134.87  11523.31     7.48 15.78      77.73
2 5309  10363.43   53523.09   6038.26     8.68 11.32      30.70
3 3713  25894.07  116185.91  32797.67    12.41 16.22      66.14
4 6294  13361.61   55596.65  16375.27    14.25 24.23      59.61
```

In *Python*, we can use `scikit-learn`'s `StandardScaler`. The `inverse_transform` method allows converting the cluster centers back to the original scale:

```
scaler = preprocessing.StandardScaler()
df0 = scaler.fit_transform(df * 1.0)

kmeans = KMeans(n_clusters=4, random_state=1).fit(df0)
counts = Counter(kmeans.labels_)

centers = pd.DataFrame(scaler.inverse_transform(kmeans.cluster_centers_),
                       columns=columns)
centers['size'] = [counts[i] for i in range(4)]
centers
```

The cluster sizes are more balanced, and the clusters are not dominated by `annual_inc` and `revol_bal`, revealing more interesting structure in the data. Note that the centers are rescaled to the original units in the preceding code. If we had left them unscaled, the resulting values would be in terms of *z*-scores and would therefore be less interpretable.

Scaling is also important for PCA. Using the *z*-scores is equivalent to using the correlation matrix (see "Correlation" on page 30) instead of the covariance matrix in computing the principal components. Software to compute PCA usually has an option to use the correlation matrix (in *R*, the `princomp` function has the argument `cor`).

Dominant Variables

Even in cases where the variables are measured on the same scale and accurately reflect relative importance (e.g., movement to stock prices), it can sometimes be useful to rescale the variables.

Suppose we add Google (GOOGL) and Amazon (AMZN) to the analysis in "Interpreting Principal Components" on page 289. We see how this is done in *R* below:

```
syms <- c('GOOGL', 'AMZN', 'AAPL', 'MSFT', 'CSCO', 'INTC', 'CVX', 'XOM',
          'SLB', 'COP', 'JPM', 'WFC', 'USB', 'AXP', 'WMT', 'TGT', 'HD', 'COST')
top_sp1 <- sp500_px[row.names(sp500_px) >= '2005-01-01', syms]
sp_pca1 <- princomp(top_sp1)
screeplot(sp_pca1)
```

In *Python*, we get the screeplot as follows:

```
syms = ['GOOGL', 'AMZN', 'AAPL', 'MSFT', 'CSCO', 'INTC', 'CVX', 'XOM',
        'SLB', 'COP', 'JPM', 'WFC', 'USB', 'AXP', 'WMT', 'TGT', 'HD', 'COST']
top_sp1 = sp500_px.loc[sp500_px.index >= '2005-01-01', syms]

sp_pca1 = PCA()
sp_pca1.fit(top_sp1)

explained_variance = pd.DataFrame(sp_pca1.explained_variance_)
ax = explained_variance.head(10).plot.bar(legend=False, figsize=(4, 4))
ax.set_xlabel('Component')
```

The screeplot displays the variances for the top principal components. In this case, the screeplot in Figure 7-13 reveals that the variances of the first and second components are much larger than the others. This often indicates that one or two variables dominate the loadings. This is, indeed, the case in this example:

```
round(sp_pca1$loadings[,1:2], 3)
      Comp.1 Comp.2
GOOGL  0.781  0.609
AMZN   0.593 -0.792
AAPL   0.078  0.004
MSFT   0.029  0.002
CSCO   0.017 -0.001
INTC   0.020 -0.001
CVX    0.068 -0.021
XOM    0.053 -0.005
...
```

In *Python*, we use the following:

```
loadings = pd.DataFrame(sp_pca1.components_[0:2, :], columns=top_sp1.columns)
loadings.transpose()
```

The first two principal components are almost completely dominated by GOOGL and AMZN. This is because the stock price movements of GOOGL and AMZN dominate the variability.

To handle this situation, you can either include them as is, rescale the variables (see "Scaling the Variables" on page 319), or exclude the dominant variables from the analysis and handle them separately. There is no "correct" approach, and the treatment depends on the application.

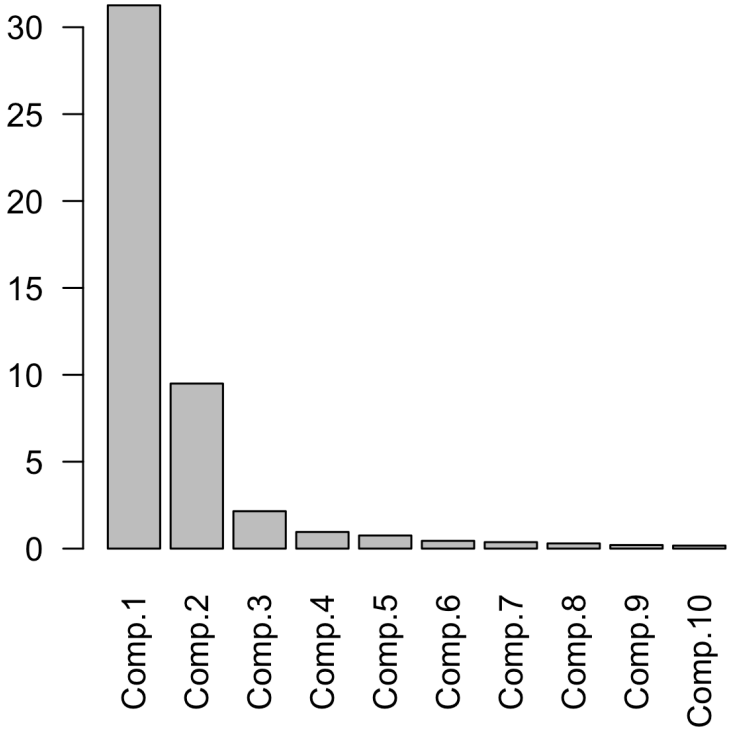

Figure 7-13. A screeplot for a PCA of top stocks from the S&P 500, including GOOGL and AMZN

Categorical Data and Gower's Distance

In the case of categorical data, you must convert it to numeric data, either by ranking (for an ordered factor) or by encoding as a set of binary (dummy) variables. If the data consists of mixed continuous and binary variables, you will usually want to scale the variables so that the ranges are similar; see "Scaling the Variables" on page 319. One popular method is to use *Gower's distance*.

The basic idea behind Gower's distance is to apply a different distance metric to each variable depending on the type of data:

- For numeric variables and ordered factors, distance is calculated as the absolute value of the difference between two records (*Manhattan distance*).

- For categorical variables, the distance is 1 if the categories between two records are different, and the distance is 0 if the categories are the same.

Gower's distance is computed as follows:

1. Compute the distance $d_{i,j}$ for all pairs of variables i and j for each record.
2. Scale each pair $d_{i,j}$ so the minimum is 0 and the maximum is 1.
3. Add the pairwise scaled distances between variables together, using either a simple or a weighted mean, to create the distance matrix.

To illustrate Gower's distance, take a few rows from the loan data in *R*:

```
> x <- loan_data[1:5, c('dti', 'payment_inc_ratio', 'home_', 'purpose_')]
> x
# A tibble: 5 × 4
    dti payment_inc_ratio   home            purpose
  <dbl>             <dbl> <fctr>             <fctr>
1  1.00           2.39320   RENT                car
2  5.55           4.57170    OWN     small_business
3 18.08           9.71600   RENT              other
4 10.08          12.21520   RENT debt_consolidation
5  7.06           3.90888   RENT              other
```

The function `daisy` in the `cluster` package in *R* can be used to compute Gower's distance:

```
library(cluster)
daisy(x, metric='gower')
Dissimilarities :
          1         2         3         4
2 0.6220479
3 0.6863877 0.8143398
4 0.6329040 0.7608561 0.4307083
5 0.3772789 0.5389727 0.3091088 0.5056250

Metric :  mixed ;  Types = I, I, N, N
Number of objects : 5
```

At the moment of this writing, Gower's distance is not available in any of the popular Python packages. However, activities are ongoing to include it in `scikit-learn`. We will update the accompanying source code once the implementation is released.

All distances are between 0 and 1. The pair of records with the biggest distance is 2 and 3: neither has the same values for `home` and `purpose`, and they have very different levels of `dti` (debt-to-income) and `payment_inc_ratio`. Records 3 and 5 have the smallest distance because they share the same values for `home` and `purpose`.

You can pass the Gower's distance matrix calculated from `daisy` to `hclust` for hierarchical clustering (see "Hierarchical Clustering" on page 304):

```
df <- defaults[sample(nrow(defaults), 250),
               c('dti', 'payment_inc_ratio', 'home', 'purpose')]
d = daisy(df, metric='gower')
hcl <- hclust(d)
dnd <- as.dendrogram(hcl)
plot(dnd, leaflab='none')
```

The resulting dendrogram is shown in Figure 7-14. The individual records are not distinguishable on the x-axis, but we can cut the dendrogram horizontally at 0.5 and examine the records in one of the subtrees with this code:

```
dnd_cut <- cut(dnd, h=0.5)
df[labels(dnd_cut$lower[[1]]),]
        dti payment_inc_ratio home_           purpose_
44532 21.22           8.37694   OWN debt_consolidation
39826 22.59           6.22827   OWN debt_consolidation
13282 31.00           9.64200   OWN debt_consolidation
31510 26.21          11.94380   OWN debt_consolidation
6693  26.96           9.45600   OWN debt_consolidation
7356  25.81           9.39257   OWN debt_consolidation
9278  21.00          14.71850   OWN debt_consolidation
13520 29.00          18.86670   OWN debt_consolidation
14668 25.75          17.53440   OWN debt_consolidation
19975 22.70          17.12170   OWN debt_consolidation
23492 22.68          18.50250   OWN debt_consolidation
```

This subtree consists entirely of owners with a loan purpose labeled as "debt_consolidation." While strict separation is not true of all subtrees, this illustrates that the categorical variables tend to be grouped together in the clusters.

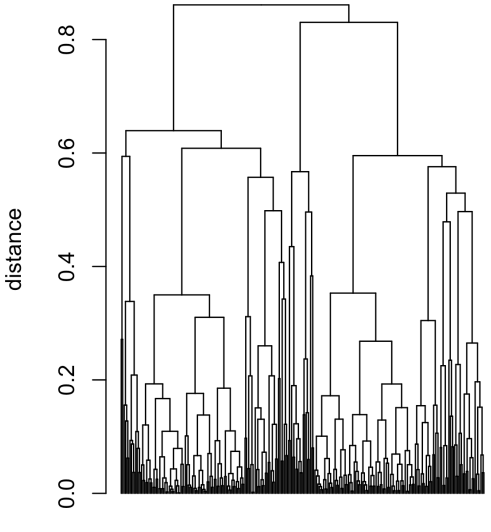

Figure 7-14. A dendrogram of `hclust` applied to a sample of loan default data with mixed variable types

Problems with Clustering Mixed Data

K-means and PCA are most appropriate for continuous variables. For smaller data sets, it is better to use hierarchical clustering with Gower's distance. In principle, there is no reason why *K*-means can't be applied to binary or categorical data. You would usually use the "one hot encoder" representation (see "One Hot Encoder" on page 242) to convert the categorical data to numeric values. In practice, however, using *K*-means and PCA with binary data can be difficult.

If the standard *z*-scores are used, the binary variables will dominate the definition of the clusters. This is because 0/1 variables take on only two values, and *K*-means can obtain a small within-cluster sum-of-squares by assigning all the records with a 0 or 1 to a single cluster. For example, apply `kmeans` to loan default data including factor variables `home` and `pub_rec_zero`, shown here in *R*:

```
df <- model.matrix(~ -1 + dti + payment_inc_ratio + home_ + pub_rec_zero,
                   data=defaults)
df0 <- scale(df)
km0 <- kmeans(df0, centers=4, nstart=10)
centers0 <- scale(km0$centers, center=FALSE,
                  scale=1/attr(df0, 'scaled:scale'))
round(scale(centers0, center=-attr(df0, 'scaled:center'), scale=FALSE), 2)
```

```
    dti payment_inc_ratio home_MORTGAGE home_OWN home_RENT pub_rec_zero
1 17.20              9.27          0.00        1      0.00         0.92
2 16.99              9.11          0.00        0      1.00         1.00
3 16.50              8.06          0.52        0      0.48         0.00
4 17.46              8.42          1.00        0      0.00         1.00
```

In *Python*:

```
columns = ['dti', 'payment_inc_ratio', 'home_', 'pub_rec_zero']
df = pd.get_dummies(defaults[columns])

scaler = preprocessing.StandardScaler()
df0 = scaler.fit_transform(df * 1.0)
kmeans = KMeans(n_clusters=4, random_state=1).fit(df0)
centers = pd.DataFrame(scaler.inverse_transform(kmeans.cluster_centers_),
                       columns=df.columns)
centers
```

The top four clusters are essentially proxies for the different levels of the factor variables. To avoid this behavior, you could scale the binary variables to have a smaller variance than other variables. Alternatively, for very large data sets, you could apply clustering to different subsets of data taking on specific categorical values. For example, you could apply clustering separately to those loans made to someone who has a mortgage, owns a home outright, or rents.

Key Ideas

- Variables measured on different scales need to be transformed to similar scales so that their impact on algorithms is not determined mainly by their scale.
- A common scaling method is normalization (standardization)—subtracting the mean and dividing by the standard deviation.
- Another method is Gower's distance, which scales all variables to the 0–1 range (it is often used with mixed numeric and categorical data).

Summary

For dimension reduction of numeric data, the main tools are either principal components analysis or *K*-means clustering. Both require attention to proper scaling of the data to ensure meaningful data reduction.

For clustering with highly structured data in which the clusters are well separated, all methods will likely produce a similar result. Each method offers its own advantage. *K*-means scales to very large data and is easily understood. Hierarchical clustering can be applied to mixed data types—numeric and categorical—and lends itself to an intuitive display (the dendrogram). Model-based clustering is founded on statistical theory and provides a more rigorous approach, as opposed to the heuristic methods. For very large data, however, *K*-means is the main method used.

With noisy data, such as the loan and stock data (and much of the data that a data scientist will face), the choice is more stark. *K*-means, hierarchical clustering, and especially model-based clustering all produce very different solutions. How should a data scientist proceed? Unfortunately, there is no simple rule of thumb to guide the choice. Ultimately, the method used will depend on the data size and the goal of the application.