7

Black Box Methods – Neural Networks and Support Vector Machines

The late science fiction author Arthur C. Clarke wrote, "any sufficiently advanced technology is indistinguishable from magic."This chapter covers a pair of machine learning methods that may appear at first glance to be magic. Though they are extremely powerful, their inner workings can be difficult to understand.

In engineering, these are referred to as **black box** processes because the mechanism that transforms the input into the output is obfuscated by an imaginary box. For instance, the black box of closed-source software intentionally conceals proprietary algorithms, the black box of political lawmaking is rooted in the bureaucratic processes, and the black box of sausage-making involves a bit of purposeful (but tasty) ignorance. In the case of machine learning, the black box is due to the complex mathematics allowing them to function.

Although they may not be easy to understand, it is dangerous to apply black box models blindly. Thus, in this chapter, we'll peek inside the box and investigate the statistical sausage-making involved in fitting such models. You'll discover:

- Neural networks mimic the structure of animal brains to model arbitrary functions
- Support vector machines use multidimensional surfaces to define the relationship between features and outcomes
- Despite their complexity, these can be applied easily to real-world problems

With any luck, you'll realize that you don't need a black belt in statistics to tackle black box machine's learning methods—there's no need to be intimidated!

Understanding neural networks

An **Artificial Neural Network** (ANN) models the relationship between a set of input signals and an output signal using a model derived from our understanding of how a biological brain responds to stimuli from sensory inputs. Just as a brain uses a network of interconnected cells called **neurons** to create a massive parallel processor, ANN uses a network of artificial neurons or **nodes** to solve learning problems.

The human brain is made up of about 85 billion neurons, resulting in a network capable of representing a tremendous amount of knowledge. As you might expect, this dwarfs the brains of other living creatures. For instance, a cat has roughly a billion neurons, a mouse has about 75 million neurons, and a cockroach has only about a million neurons. In contrast, many ANNs contain far fewer neurons, typically only several hundred, so we're in no danger of creating an artificial brain anytime in the near future—even a fruit fly brain with 100,000 neurons far exceeds the current state-of-the-art ANN.

Though it may be unfeasible to completely model a cockroach's brain, a neural network may still provide an adequate heuristic model of its behavior. Suppose that we develop an algorithm that can mimic how a roach flees when discovered. If the behavior of the robot roach is convincing, does it matter whether its brain is as sophisticated as the living creature's? This question is the basis of the controversial **Turing test**, proposed in 1950 by the pioneering computer scientist Alan Turing, proposed in 1950 by the pioneering computer scientist Alan Turing, which grades a machine as intelligent if a human being cannot distinguish its behavior from a living creature's.

Rudimentary ANNs have been used for over 50 years to simulate the brain's approach to problem-solving. At first, this involved learning simple functions like the logical AND function or the logical OR function. These early exercises were used primarily to help scientists understand how biological brains might operate. However, as computers have become increasingly powerful in the recent years, the complexity of ANNs has likewise increased so much that they are now frequently applied to more practical problems including:

- Speech and handwriting recognition programs like those used by voicemail transcription services and postal mail sorting machines
- The automation of smart devices like an office building's environmental controls or self-driving cars and self-piloting drones
- Sophisticated models of weather and climate patterns, tensile strength, fluid dynamics, and many other scientific, social, or economic phenomena

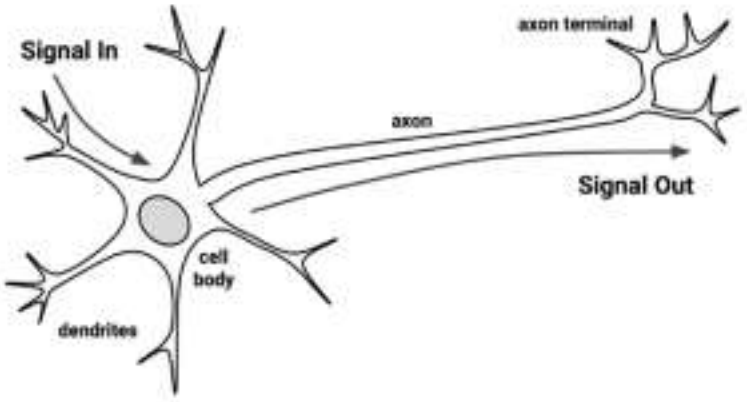
Broadly speaking, ANNs are versatile learners that can be applied to nearly any learning task: classification, numeric prediction, and even unsupervised pattern recognition.

Whether deserving or not, ANN learners are often reported in the media with great fanfare. For instance, an "*artificial brain*" developed by Google was recently touted for its ability to identify cat videos on YouTube. Such hype may have less to do with anything unique to ANNs and more to do with the fact that ANNs are captivating because of their similarities to living minds.

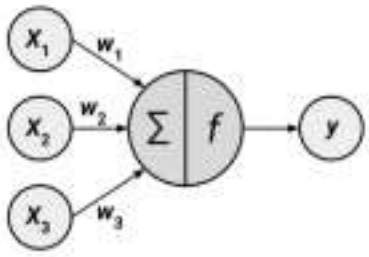
ANNs are best applied to problems where the input data and output data are well-defined or at least fairly simple, yet the process that relates the input to output is extremely complex. As a black box method, they work well for these types of black box problems.

From biological to artificial neurons

Because ANNs were intentionally designed as conceptual models of human brain activity, it is helpful to first understand how biological neurons function. As illustrated in the following figure, incoming signals are received by the cell's **dendrites** through a biochemical process. The process allows the impulse to be weighted according to its relative importance or frequency. As the **cell body** begins accumulating the incoming signals, a threshold is reached at which the cell fires and the output signal is transmitted via an electrochemical process down the **axon**. At the axon's terminals, the electric signal is again processed as a chemical signal to be passed to the neighboring neurons across a tiny gap known as a **synapse**.



The model of a single artificial neuron can be understood in terms very similar to the biological model. As depicted in the following figure, a directed network diagram defines a relationship between the input signals received by the dendrites (x variables), and the output signal (y variable). Just as with the biological neuron, each dendrite's signal is weighted (w values) according to its importance—ignore, for now, how these weights are determined. The input signals are summed by the cell body and the signal is passed on according to an **activation function** denoted by f:



A typical artificial neuron with n input dendrites can be represented by the formula that follows. The w weights allow each of the n inputs (denoted by x_i) to contribute a greater or lesser amount to the sum of input signals. The net total is used by the activation function $f(x)$, and the resulting signal, $y(x)$, is the output axon:

$$y(x) = f\left(\sum_{i=1}^{n} w_i x_i\right)$$

Neural networks use neurons defined this way as building blocks to construct complex models of data. Although there are numerous variants of neural networks, each can be defined in terms of the following characteristics:

- An **activation function**, which transforms a neuron's combined input signals into a single output signal to be broadcasted further in the network
- A **network topology** (or architecture), which describes the number of neurons in the model as well as the number of layers and manner in which they are connected
- The **training algorithm** that specifies how connection weights are set in order to inhibit or excite neurons in proportion to the input signal

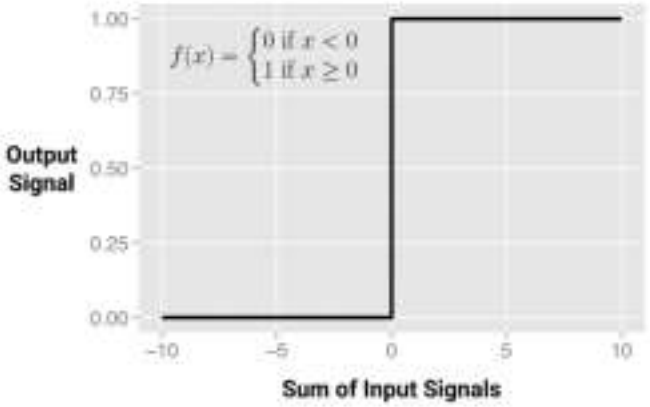
Let's take a look at some of the variations within each of these categories to see how they can be used to construct typical neural network models.

Activation functions

The activation function is the mechanism by which the artificial neuron processes incoming information and passes it throughout the network. Just as the artificial neuron is modeled after the biological version, so is the activation function modeled after nature's design.

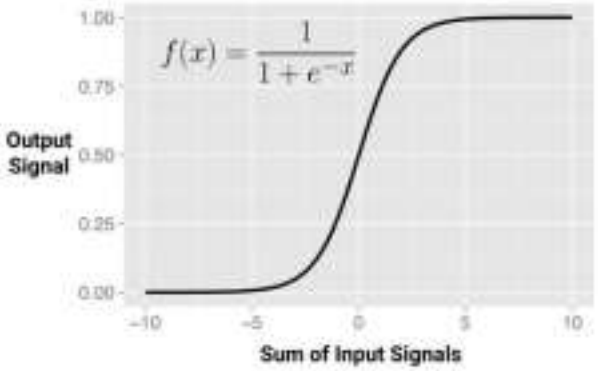
In the biological case, the activation function could be imagined as a process that involves summing the total input signal and determining whether it meets the firing threshold. If so, the neuron passes on the signal; otherwise, it does nothing. In ANN terms, this is known as a **threshold activation function**, as it results in an output signal only once a specified input threshold has been attained.

The following figure depicts a typical threshold function; in this case, the neuron fires when the sum of input signals is at least zero. Because its shape resembles a stair, it is sometimes called a **unit step activation function**.

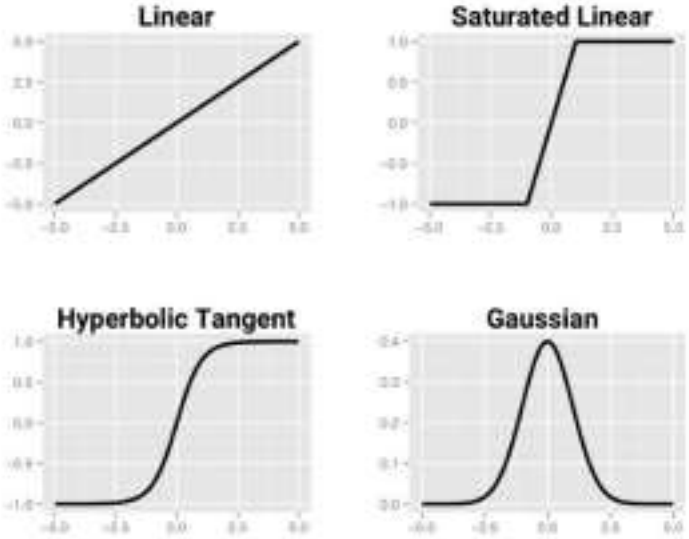


Although the threshold activation function is interesting due to its parallels with biology, it is rarely used in artificial neural networks. Freed from the limitations of biochemistry, the ANN activation functions can be chosen based on their ability to demonstrate desirable mathematical characteristics and accurately model relationships among data.

Perhaps the most commonly used alternative is the **sigmoid activation function** (more specifically, the *logistic* sigmoid) shown in the following figure. Note that in the formula shown, e is the base of the natural logarithm (approximately 2.72). Although it shares a similar step or "S" shape with the threshold activation function, the output signal is no longer binary; output values can fall anywhere in the range from 0 to 1. Additionally, the sigmoid is **differentiable**, which means that it is possible to calculate the derivative across the entire range of inputs. As you will learn later, this feature is crucial to create efficient ANN optimization algorithms.



Although sigmoid is perhaps the most commonly used activation function and is often used by default, some neural network algorithms allow a choice of alternatives. A selection of such activation functions is shown in the following figure:



The primary detail that differentiates these activation functions is the output signal range. Typically, this is one of (0, 1), (-1, +1), or (-inf, +inf). The choice of activation function biases the neural network such that it may fit certain types of data more appropriately, allowing the construction of specialized neural networks. For instance, a linear activation function results in a neural network very similar to a linear regression model, while a Gaussian activation function results in a model called a **Radial Basis Function (RBF)** network. Each of these has strengths better suited for certain learning tasks and not others.

It's important to recognize that for many of the activation functions, the range of input values that affect the output signal is relatively narrow. For example, in the case of sigmoid, the output signal is always nearly 0 or 1 for an input signal below -5 or above +5, respectively. The compression of signal in this way results in a saturated signal at the high and low ends of very dynamic inputs, just as turning a guitar amplifier up too high results in a distorted sound due to clipping of the peaks of sound waves. Because this essentially squeezes the input values into a smaller range of outputs, activation functions like the sigmoid are sometimes called **squashing functions**.

The solution to the squashing problem is to transform all neural network inputs such that the features' values fall within a small range around 0. Typically, this involves standardizing or normalizing the features. By restricting the range of input values, the activation function will have action across the entire range, preventing large-valued features such as household income from dominating small-valued features such as the number of children in the household. A side benefit is that the model may also be faster to train, since the algorithm can iterate more quickly through the actionable range of input values.

Although theoretically a neural network can adapt to a very dynamic feature by adjusting its weight over many iterations. In extreme cases, many algorithms will stop iterating long before this occurs. If your model is making predictions that do not make sense, double-check whether you've correctly standardized the input data.

Network topology

The ability of a neural network to learn is rooted in its **topology**, or the patterns and structures of interconnected neurons. Although there are countless forms of network architecture, they can be differentiated by three key characteristics:

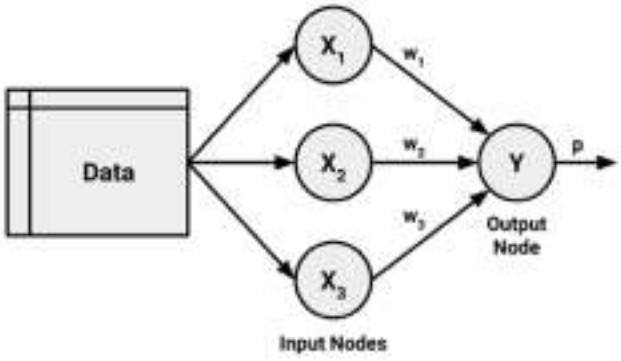
- The number of layers
- Whether information in the network is allowed to travel backward
- The number of nodes within each layer of the network

The topology determines the complexity of tasks that can be learned by the network. Generally, larger and more complex networks are capable of identifying more subtle patterns and complex decision boundaries. However, the power of a network is not only a function of the network size, but also the way units are arranged.

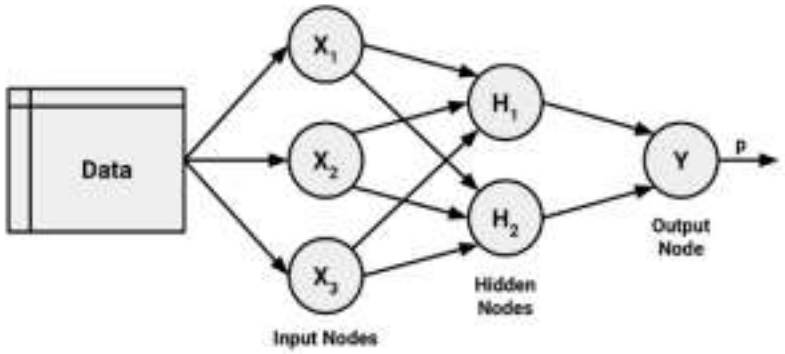
The number of layers

To define topology, we need a terminology that distinguishes artificial neurons based on their position in the network. The figure that follows illustrates the topology of a very simple network. A set of neurons called **input nodes** receives unprocessed signals directly from the input data. Each input node is responsible for processing a single feature in the dataset; the feature's value will be transformed by the corresponding node's activation function. The signals sent by the input nodes are received by the output node, which uses its own activation function to generate a final prediction (denoted here as *p*).

The input and output nodes are arranged in groups known as **layers**. Because the input nodes process the incoming data exactly as it is received, the network has only one set of connection weights (labeled here as w_1, w_2, and w_3). It is therefore termed a **single-layer network**. Single-layer networks can be used for basic pattern classification, particularly for patterns that are linearly separable, but more sophisticated networks are required for most learning tasks.



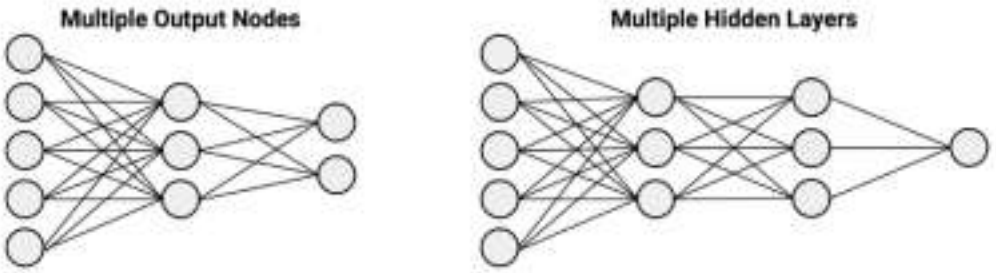
As you might expect, an obvious way to create more complex networks is by adding additional layers. As depicted here, a **multilayer network** adds one or more **hidden layers** that process the signals from the input nodes prior to it reaching the output node. Most multilayer networks are **fully connected**, which means that every node in one layer is connected to every node in the next layer, but this is not required.



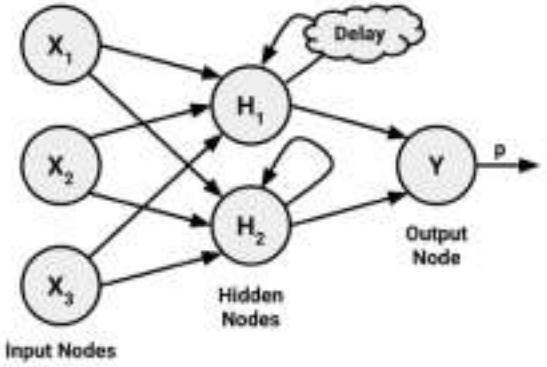
The direction of information travel

You may have noticed that in the prior examples, arrowheads were used to indicate signals traveling in only one direction. Networks in which the input signal is fed continuously in one direction from connection to connection until it reaches the output layer are called **feedforward** networks.

In spite of the restriction on information flow, feedforward networks offer a surprising amount of flexibility. For instance, the number of levels and nodes at each level can be varied, multiple outcomes can be modeled simultaneously, or multiple hidden layers can be applied. A neural network with multiple hidden layers is called a **Deep Neural Network** (**DNN**) and the practice of training such network is sometimes referred to as **deep learning**.



In contrast, a **recurrent network** (or **feedback network**) allows signals to travel in both directions using loops. This property, which more closely mirrors how a biological neural network works, allows extremely complex patterns to be learned. The addition of a short-term memory, or **delay**, increases the power of recurrent networks immensely. Notably, this includes the capability to understand the sequences of events over a period of time. This could be used for stock market prediction, speech comprehension, or weather forecasting. A simple recurrent network is depicted as follows:



In spite of their potential, recurrent networks are still largely theoretical and are rarely used in practice. On the other hand, feedforward networks have been extensively applied to real-world problems. In fact, the multilayer feedforward network, sometimes called the **Multilayer Perceptron** (**MLP**), is the de facto standard ANN topology. If someone mentions that they are fitting a neural network, they are most likely referring to a MLP.

The number of nodes in each layer

In addition to the variations in the number of layers and the direction of information travel, neural networks can also vary in complexity by the number of nodes in each layer. The number of input nodes is predetermined by the number of features in the input data. Similarly, the number of output nodes is predetermined by the number of outcomes to be modeled or the number of class levels in the outcome. However, the number of hidden nodes is left to the user to decide prior to training the model.

Unfortunately, there is no reliable rule to determine the number of neurons in the hidden layer. The appropriate number depends on the number of input nodes, the amount of training data, the amount of noisy data, and the complexity of the learning task, among many other factors.

In general, more complex network topologies with a greater number of network connections allow the learning of more complex problems. A greater number of neurons will result in a model that more closely mirrors the training data, but this runs a risk of overfitting; it may generalize poorly to future data. Large neural networks can also be computationally expensive and slow to train.

The best practice is to use the fewest nodes that result in adequate performance in a validation dataset. In most cases, even with only a small number of hidden nodes—often as few as a handful—the neural network can offer a tremendous amount of learning ability.

It has been proven that a neural network with at least one hidden layer of sufficient neurons is a **universal function approximator**. This means that neural networks can be used to approximate any continuous function to an arbitrary precision over a finite interval.

Training neural networks with backpropagation

The network topology is a blank slate that by itself has not learned anything. Like a newborn child, it must be trained with experience. As the neural network processes the input data, connections between the neurons are strengthened or weakened, similar to how a baby's brain develops as he or she experiences the environment. The network's connection weights are adjusted to reflect the patterns observed over time.

Training a neural network by adjusting connection weights is very computationally intensive. Consequently, though they had been studied for decades prior, ANNs were rarely applied to real-world learning tasks until the mid-to-late 1980s, when an efficient method of training an ANN was discovered. The algorithm, which used a strategy of back-propagating errors, is now known simply as **backpropagation**.

Coincidentally, several research teams independently discovered and published the backpropagation algorithm around the same time. Among them, perhaps the most often cited work is: Rumelhart DE, Hinton GE, Williams RJ. Learning representations by back-propagating errors. *Nature*. 1986; 323:533-566.

Although still notoriously slow relative to many other machine learning algorithms, the backpropagation method led to a resurgence of interest in ANNs. As a result, multilayer feedforward networks that use the backpropagation algorithm are now common in the field of data mining. Such models offer the following strengths and weaknesses:

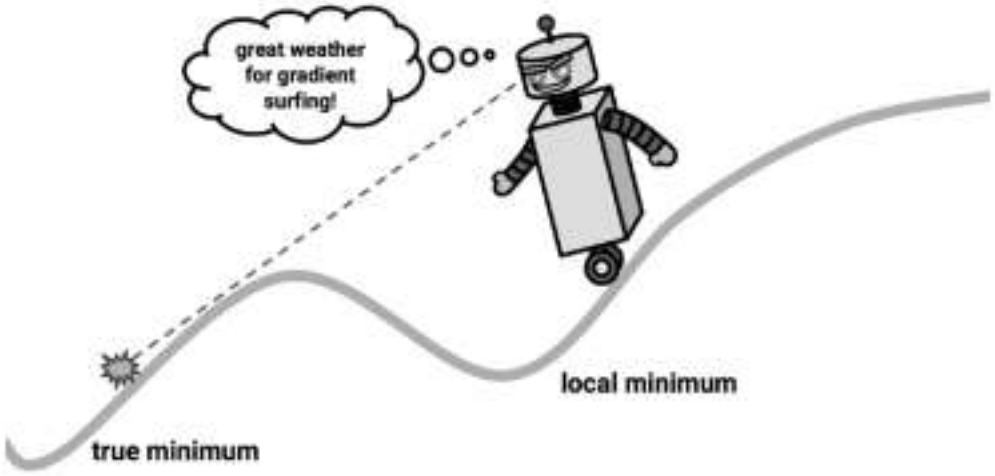
Strengths	Weaknesses
• Can be adapted to classification or numeric prediction problems • Capable of modeling more complex patterns than nearly any algorithm • Makes few assumptions about the data's underlying relationships	• Extremely computationally intensive and slow to train, particularly if the network topology is complex • Very prone to overfitting training data • Results in a complex black box model that is difficult, if not impossible, to interpret

In its most general form, the backpropagation algorithm iterates through many cycles of two processes. Each cycle is known as an **epoch**. Because the network contains no *a priori* (existing) knowledge, the starting weights are typically set at random. Then, the algorithm iterates through the processes, until a stopping criterion is reached. Each epoch in the backpropagation algorithm includes:

- A **forward phase** in which the neurons are activated in sequence from the input layer to the output layer, applying each neuron's weights and activation function along the way. Upon reaching the final layer, an output signal is produced.
- A **backward phase** in which the network's output signal resulting from the forward phase is compared to the true target value in the training data. The difference between the network's output signal and the true value results in an error that is propagated backwards in the network to modify the connection weights between neurons and reduce future errors.

Over time, the network uses the information sent backward to reduce the total error of the network. Yet one question remains: because the relationship between each neuron's inputs and outputs is complex, how does the algorithm determine how much a weight should be changed? The answer to this question involves a technique called **gradient descent**. Conceptually, it works similarly to how an explorer trapped in the jungle might find a path to water. By examining the terrain and continually walking in the direction with the greatest downward slope, the explorer will eventually reach the lowest valley, which is likely to be a riverbed.

In a similar process, the backpropagation algorithm uses the derivative of each neuron's activation function to identify the gradient in the direction of each of the incoming weights—hence the importance of having a differentiable activation function. The gradient suggests how steeply the error will be reduced or increased for a change in the weight. The algorithm will attempt to change the weights that result in the greatest reduction in error by an amount known as the **learning rate**. The greater the learning rate, the faster the algorithm will attempt to descend down the gradients, which could reduce the training time at the risk of overshooting the valley.



Although this process seems complex, it is easy to apply in practice. Let's apply our understanding of multilayer feedforward networks to a real-world problem.

Example – Modeling the strength of concrete with ANNs

In the field of engineering, it is crucial to have accurate estimates of the performance of building materials. These estimates are required in order to develop safety guidelines governing the materials used in the construction of buildings, bridges, and roadways.

Estimating the strength of concrete is a challenge of particular interest. Although it is used in nearly every construction project, concrete performance varies greatly due to a wide variety of ingredients that interact in complex ways. As a result, it is difficult to accurately predict the strength of the final product. A model that could reliably predict concrete strength given a listing of the composition of the input materials could result in safer construction practices.

Step 1 – collecting data

For this analysis, we will utilize data on the compressive strength of concrete donated to the UCI Machine Learning Data Repository (`http://archive.ics.uci.edu/ml`) by I-Cheng Yeh. As he found success using neural networks to model these data, we will attempt to replicate his work using a simple neural network model in R.

For more information on Yeh's approach to this learning task, refer to: Yeh IC. Modeling of strength of high performance concrete using artificial neural networks. *Cement and Concrete Research*. 1998; 28:1797-1808.

According to the website, the concrete dataset contains 1,030 examples of concrete with eight features describing the components used in the mixture. These features are thought to be related to the final compressive strength and they include the amount (in kilograms per cubic meter) of cement, slag, ash, water, superplasticizer, coarse aggregate, and fine aggregate used in the product in addition to the aging time (measured in days).

To follow along with this example, download the `concrete.csv` file from the Packt Publishing website and save it to your R working directory.

Step 2 – exploring and preparing the data

As usual, we'll begin our analysis by loading the data into an R object using the `read.csv()` function, and confirming that it matches the expected structure:

```
> concrete <- read.csv("concrete.csv")
> str(concrete)
'data.frame':   1030 obs. of  9 variables:
 $ cement      : num  141 169 250 266 155 ...
 $ slag        : num  212 42.2 0 114 183.4 ...
 $ ash         : num  0 124.3 95.7 0 0 ...
```

```
$ water       : num  204 158 187 228 193 ...
$ superplastic: num  0 10.8 5.5 0 9.1 0 0 6.4 0 9 ...
$ coarseagg   : num  972 1081 957 932 1047 ...
$ fineagg     : num  748 796 861 670 697 ...
$ age         : int  28 14 28 28 28 90 7 56 28 28 ...
$ strength    : num  29.9 23.5 29.2 45.9 18.3 ...
```

The nine variables in the data frame correspond to the eight features and one outcome we expected, although a problem has become apparent. Neural networks work best when the input data are scaled to a narrow range around zero, and here, we see values ranging anywhere from zero up to over a thousand.

Typically, the solution to this problem is to rescale the data with a normalizing or standardization function. If the data follow a bell-shaped curve (a normal distribution as described in *Chapter 2, Managing and Understanding Data*), then it may make sense to use standardization via R's built-in scale() function. On the other hand, if the data follow a uniform distribution or are severely nonnormal, then normalization to a 0-1 range may be more appropriate. In this case, we'll use the latter.

In *Chapter 3, Lazy Learning – Classification Using Nearest Neighbors*, we defined our own `normalize()` function as:

```
> normalize <- function(x) {
    return((x - min(x)) / (max(x) - min(x)))
  }
```

After executing this code, our `normalize()` function can be applied to every column in the concrete data frame using the `lapply()` function as follows:

```
> concrete_norm <- as.data.frame(lapply(concrete, normalize))
```

To confirm that the normalization worked, we can see that the minimum and maximum strength are now 0 and 1, respectively:

```
> summary(concrete_norm$strength)
    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.0000  0.2664  0.4001  0.4172  0.5457  1.0000
```

In comparison, the original minimum and maximum values were 2.33 and 82.60:

```
> summary(concrete$strength)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   2.33   23.71   34.44   35.82   46.14   82.60
```

Any transformation applied to the data prior to training the model will have to be applied in reverse later on, in order to convert back to the original units of measurement. To facilitate the rescaling, it is wise to save the original data or at least the summary statistics of the original data.

Following Yeh's precedent in the original publication, we will partition the data into a training set with 75 percent of the examples and a testing set with 25 percent. The CSV file we used was already sorted in random order, so we simply need to divide it into two portions:

```
> concrete_train <- concrete_norm[1:773, ]
> concrete_test <- concrete_norm[774:1030, ]
```

We'll use the training dataset to build the neural network and the testing dataset to evaluate how well the model generalizes to future results. As it is easy to overfit a neural network, this step is very important.

Step 3 – training a model on the data

To model the relationship between the ingredients used in concrete and the strength of the finished product, we will use a multilayer feedforward neural network. The `neuralnet` package by Stefan Fritsch and Frauke Guenther provides a standard and easy-to-use implementation of such networks. It also offers a function to plot the network topology. For these reasons, the `neuralnet` implementation is a strong choice for learning more about neural networks, though this is not to say that it cannot be used to accomplish real work as well—it's quite a powerful tool, as you will soon see.

There are several other commonly used packages to train ANN models in R, each with unique strengths and weaknesses. Because it ships as a part of the standard R installation, the `nnet` package is perhaps the most frequently cited ANN implementation. It uses a slightly more sophisticated algorithm than standard backpropagation. Another strong option is the `RSNNS` package, which offers a complete suite of neural network functionality with the downside being that it is more difficult to learn.

As `neuralnet` is not included in base R, you will need to install it by typing `install.packages("neuralnet")` and load it with the `library(neuralnet)` command. The included `neuralnet()` function can be used for training neural networks for numeric prediction using the following syntax:

Neural network syntax

using the `neuralnet()` function in the `neuralnet` package

Building the model:

```
m <- neuralnet(target ~ predictors, data = mydata,
               hidden = 1)
```

- `target` is the outcome in the `mydata` data frame to be modeled
- `predictors` is an R formula specifying the features in the `mydata` data frame to use for prediction
- `data` specifies the data frame in which the `target` and `predictors` variables can be found
- `hidden` specifies the number of neurons in the hidden layer (by default, 1)

The function will return a neural network object that can be used to make predictions.

Making predictions:

```
p <- compute(m, test)
```

- `m` is a model trained by the `neuralnet()` function
- `test` is a data frame containing test data with the same features as the training data used to build the classifier

The function will return a list with two components: `$neurons`, which stores the neurons for each layer in the network, and `$net.result`, which stores the model's predicted values.

Example:

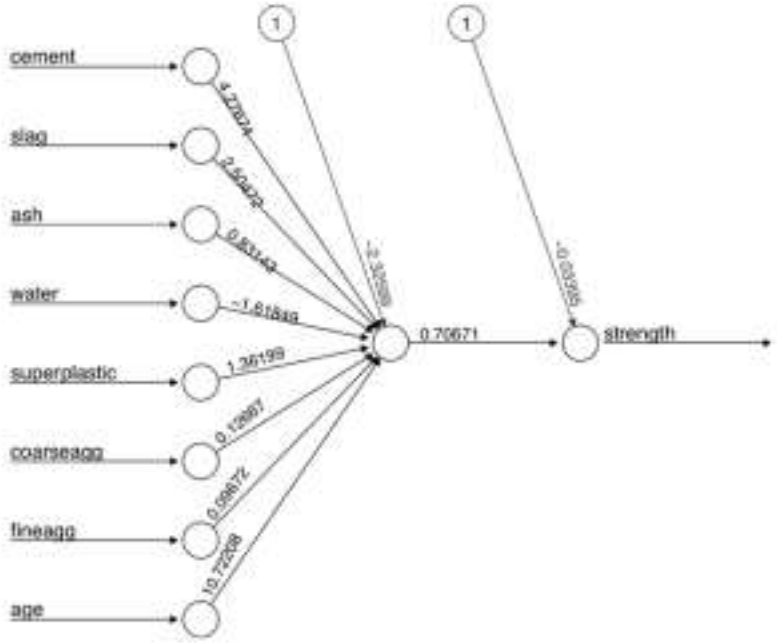
```
concrete_model <- neuralnet(strength ~ cement + slag
  + ash, data = concrete)
model_results <- compute(concrete_model,
  concrete_data)
strength_predictions <- model_results$net.result
```

We'll begin by training the simplest multilayer feedforward network with only a single hidden node:

```
> concrete_model <- neuralnet(strength ~ cement + slag
  + ash + water + superplastic + coarseagg + fineagg + age,
  data = concrete_train)
```

We can then visualize the network topology using the `plot()` function on the resulting model object:

```
> plot(concrete_model)
```



Error: 5.077438 Steps: 4882

In this simple model, there is one input node for each of the eight features, followed by a single hidden node and a single output node that predicts the concrete strength. The weights for each of the connections are also depicted, as are the **bias terms** (indicated by the nodes labeled with the number **1**). The bias terms are numeric constants that allow the value at the indicated nodes to be shifted upward or downward, much like the intercept in a linear equation.

A neural network with a single hidden node can be thought of as a distant cousin of the linear regression models we studied in *Chapter 6, Forecasting Numeric Data – Regression Methods*. The weight between each input node and the hidden node is similar to the regression coefficients, and the weight for the bias term is similar to the intercept.

At the bottom of the figure, R reports the number of training steps and an error measure called the **Sum of Squared Errors** (**SSE**), which as you might expect, is the sum of the squared predicted minus actual values. A lower SSE implies better predictive performance. This is helpful for estimating the model's performance on the training data, but tells us little about how it will perform on unseen data.

Step 4 – evaluating model performance

The network topology diagram gives us a peek into the black box of the ANN, but it doesn't provide much information about how well the model fits future data. To generate predictions on the test dataset, we can use the `compute()` as follows:

```
> model_results <- compute(concrete_model, concrete_test[1:8])
```

The `compute()` function works a bit differently from the `predict()` functions we've used so far. It returns a list with two components: `$neurons`, which stores the neurons for each layer in the network, and `$net.result`, which stores the predicted values. We'll want the latter:

```
> predicted_strength <- model_results$net.result
```

Because this is a numeric prediction problem rather than a classification problem, we cannot use a confusion matrix to examine model accuracy. Instead, we must measure the correlation between our predicted concrete strength and the true value. This provides insight into the strength of the linear association between the two variables.

Recall that the `cor()` function is used to obtain a correlation between two numeric vectors:

```
> cor(predicted_strength, concrete_test$strength)
             [,1]
[1,] 0.8064655576
```

Don't be alarmed if your result differs. Because the neural network begins with random weights, the predictions can vary from model to model. If you'd like to match these results exactly, try using `set.seed(12345)` before building the neural network.

Correlations close to 1 indicate strong linear relationships between two variables. Therefore, the correlation here of about 0.806 indicates a fairly strong relationship. This implies that our model is doing a fairly good job, even with only a single hidden node.

Given that we only used one hidden node, it is likely that we can improve the performance of our model. Let's try to do a bit better.

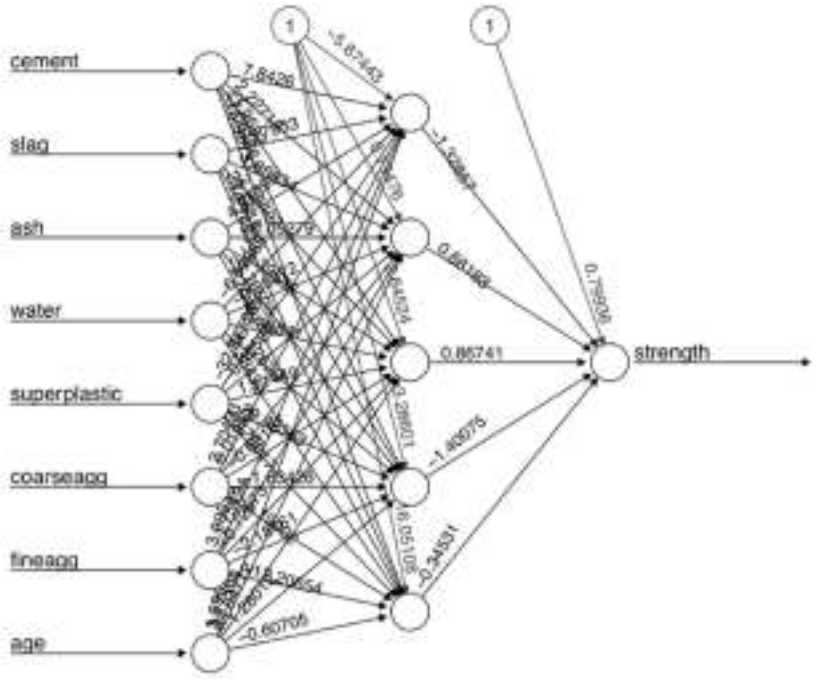
Step 5 – improving model performance

As networks with more complex topologies are capable of learning more difficult concepts, let's see what happens when we increase the number of hidden nodes to five. We use the `neuralnet()` function as before, but add the `hidden = 5` parameter:

```
> concrete_model2 <- neuralnet(strength ~ cement + slag +
                               ash + water + superplastic +
                               coarseagg + fineagg + age,
                               data = concrete_train, hidden = 5)
```

Plotting the network again, we see a drastic increase in the number of connections. We can see how this impacted the performance as follows:

```
> plot(concrete_model2)
```



Error: 1.626684 Steps: 86849

Notice that the reported error (measured again by SSE) has been reduced from 5.08 in the previous model to 1.63 here. Additionally, the number of training steps rose from 4,882 to 86,849, which should come as no surprise given how much more complex the model has become. More complex networks take many more iterations to find the optimal weights.

Applying the same steps to compare the predicted values to the true values, we now obtain a correlation around 0.92, which is a considerable improvement over the previous result of 0.80 with a single hidden node:

```
> model_results2 <- compute(concrete_model2, concrete_test[1:8])
> predicted_strength2 <- model_results2$net.result
> cor(predicted_strength2, concrete_test$strength)
             [,1]
[1,] 0.9244533426
```

Interestingly, in the original publication, Yeh reported a mean correlation of 0.885 using a very similar neural network. This means that with relatively little effort, we were able to match the performance of a subject-matter expert. If you'd like more practice with neural networks, you might try applying the principles learned earlier in this chapter to see how it impacts model performance. Perhaps try using different numbers of hidden nodes, applying different activation functions, and so on. The `?neuralnet` help page provides more information on the various parameters that can be adjusted.

Understanding Support Vector Machines

A **Support Vector Machine** (**SVM**) can be imagined as a surface that creates a boundary between points of data plotted in multidimensional that represent examples and their feature values. The goal of a SVM is to create a flat boundary called a **hyperplane**, which divides the space to create fairly homogeneous partitions on either side. In this way, the SVM learning combines aspects of both the instance-based nearest neighbor learning presented in *Chapter 3, Lazy Learning – Classification Using Nearest Neighbors*, and the linear regression modeling described in *Chapter 6, Forecasting Numeric Data – Regression Methods*. The combination is extremely powerful, allowing SVMs to model highly complex relationships.

Although the basic mathematics that drive SVMs have been around for decades, they have recently exploded in popularity. This is, of course, rooted in their state-of-the-art performance, but perhaps also due to the fact that award winning SVM algorithms have been implemented in several popular and well-supported libraries across many programming languages, including R. SVMs have thus been adopted by a much wider audience, might have otherwise been unable to apply the somewhat complex math needed to implement a SVM. The good news is that although the math may be difficult, the basic concepts are understandable.

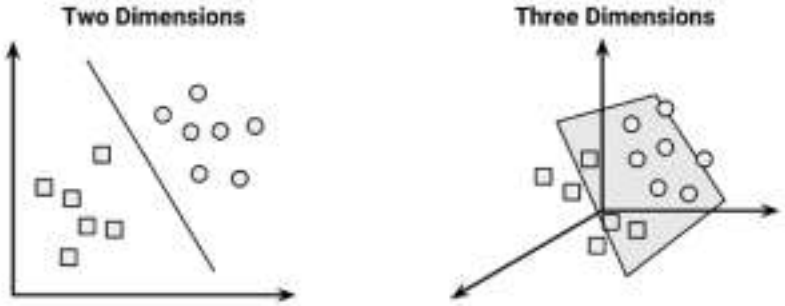
SVMs can be adapted for use with nearly any type of learning task, including both classification and numeric prediction. Many of the algorithm's key successes have come in pattern recognition. Notable applications include:

- Classification of microarray gene expression data in the field of bioinformatics to identify cancer or other genetic diseases
- Text categorization such as identification of the language used in a document or the classification of documents by subject matter
- The detection of rare yet important events like combustion engine failure, security breaches, or earthquakes

SVMs are most easily understood when used for binary classification, which is how the method has been traditionally applied. Therefore, in the remaining sections, we will focus only on SVM classifiers. Don't worry, however, as the same principles you learn here will apply while adapting SVMs to other learning tasks such as numeric prediction.

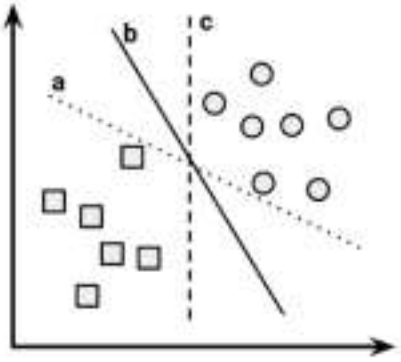
Classification with hyperplanes

As noted previously, SVMs use a boundary called a hyperplane to partition data into groups of similar class values. For example, the following figure depicts hyperplanes that separate groups of circles and squares in two and three dimensions. Because the circles and squares can be separated perfectly by the straight line or flat surface, they are said to be **linearly separable**. At first, we'll consider only the simple case where this is true, but SVMs can also be extended to problems where the points are not linearly separable.



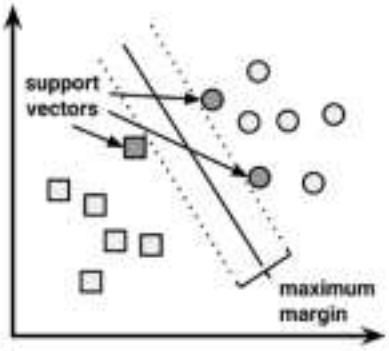
For convenience, the hyperplane is traditionally depicted as a line in 2D space, but this is simply because it is difficult to illustrate space in greater than two dimensions. In reality, the hyperplane is a flat surface in a high-dimensional space—a concept that can be difficult to get your mind around.

In two dimensions, the task of the SVM algorithm is to identify a line that separates the two classes. As shown in the following figure, there is more than one choice of dividing line between the groups of circles and squares. Three such possibilities are labeled **a**, **b**, and **c**. How does the algorithm choose?



The answer to that question involves a search for the **Maximum Margin Hyperplane** (**MMH**) that creates the greatest separation between the two classes. Although any of the three lines separating the circles and squares would correctly classify all the data points, it is likely that the line that leads to the greatest separation will generalize the best to the future data. The maximum margin will improve the chance that, in spite of random noise, the points will remain on the correct side of the boundary.

The **support vectors** (indicated by arrows in the figure that follows) are the points from each class that are the closest to the MMH; each class must have at least one support vector, but it is possible to have more than one. Using the support vectors alone, it is possible to define the MMH. This is a key feature of SVMs; the support vectors provide a very compact way to store a classification model, even if the number of features is extremely large.

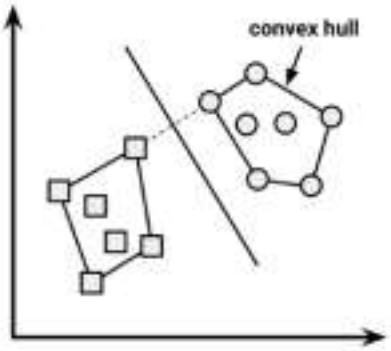


The algorithm to identify the support vectors relies on vector geometry and involves some fairly tricky math that is outside the scope of this book. However, the basic principles of the process are fairly straightforward.

More information on the mathematics of SVMs can be found in the classic paper: Cortes C, Vapnik V. Support-vector network. *Machine Learning*. 1995; 20:273-297. A beginner level discussion can be found in: Bennett KP, Campbell C. Support vector machines: hype or hallelujah. SIGKDD Explorations. 2003; 2:1-13. A more in-depth look can be found in: Steinwart I, Christmann A. *Support Vector Machines*. New York: Springer; 2008.

The case of linearly separable data

It is easiest to understand how to find the maximum margin under the assumption that the classes are linearly separable. In this case, the MMH is as far away as possible from the outer boundaries of the two groups of data points. These outer boundaries are known as the **convex hull**. The MMH is then the perpendicular bisector of the shortest line between the two convex hulls. Sophisticated computer algorithms that use a technique known as **quadratic optimization** are capable of finding the maximum margin in this way.



An alternative (but equivalent) approach involves a search through the space of every possible hyperplane in order to find a set of two parallel planes that divide the points into homogeneous groups yet themselves are as far apart as possible. To use a metaphor, one can imagine this process as similar to trying to find the thickest mattress that can fit up a stairwell to your bedroom.

To understand this search process, we'll need to define exactly what we mean by a hyperplane. In *n*-dimensional space, the following equation is used:

$$\vec{w} \cdot \vec{x} + b = 0$$

If you aren't familiar with this notation, the arrows above the letters indicate that they are vectors rather than single numbers. In particular, w is a vector of n weights, that is, $\{w_1, w_2, ..., w_n\}$, and b is a single number known as the **bias**. The bias is conceptually equivalent to the intercept term in the slope-intercept form discussed in *Chapter 6, Forecasting Numeric Data – Regression Methods*.

If you're having trouble imagining the plane, don't worry about the details. Simply think of the equation as a way to specify a surface, much like when the slope-intercept form ($y = mx + b$) is used to specify lines in 2D space.

Using this formula, the goal of the process is to find a set of weights that specify two hyperplanes, as follows:

$$\vec{w} \cdot \vec{x} + b \geq +1$$
$$\vec{w} \cdot \vec{x} + b \leq -1$$

We will also require that these hyperplanes are specified such that all the points of one class fall above the first hyperplane and all the points of the other class fall beneath the second hyperplane. This is possible so long as the data are linearly separable.

Vector geometry defines the distance between these two planes as:

$$\frac{2}{||\vec{w}||}$$

Here, $||w||$ indicates the **Euclidean norm** (the distance from the origin to vector w). Because $||w||$ is in the denominator, to maximize distance, we need to minimize $||w||$. The task is typically reexpressed as a set of constraints, as follows:

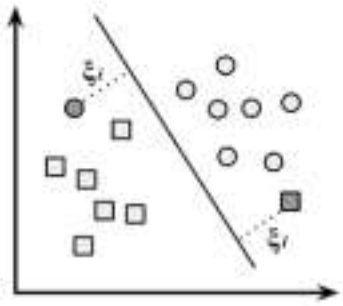
$$\min \frac{1}{2} \|\vec{w}\|^2$$
$$s.t.\ \ y_i(\vec{w} \cdot \vec{x}_i - b) \geq 1, \forall \vec{x}_i$$

Although this looks messy, it's really not too complicated to understand conceptually. Basically, the first line implies that we need to minimize the Euclidean norm (squared and divided by two to make the calculation easier). The second line notes that this is subject to (*s.t.*), the condition that each of the y_i data points is correctly classified. Note that y indicates the class value (transformed to either +1 or -1) and the upside down "A" is shorthand for "for all."

As with the other method for finding the maximum margin, finding a solution to this problem is a task best left for quadratic optimization software. Although it can be processor-intensive, specialized algorithms are capable of solving these problems quickly even on fairly large datasets.

The case of nonlinearly separable data

As we've worked through the theory behind SVMs, you may be wondering about the elephant in the room: what happens if the data are not linearly separable? The solution to this problem is the use of a **slack variable**, which creates a soft margin that allows some points to fall on the incorrect side of the margin. The figure that follows illustrates two points falling on the wrong side of the line with the corresponding slack terms (denoted with the Greek letter Xi):



A cost value (denoted as *C*) is applied to all points that violate the constraints, and rather than finding the maximum margin, the algorithm attempts to minimize the total cost. We can therefore revise the optimization problem to:

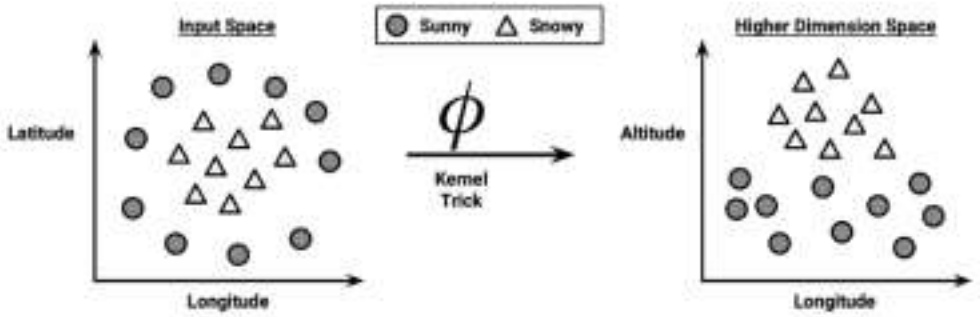
$$\min \frac{1}{2}\|\vec{w}\|^2 + C\sum_{i=1}^{n}\xi_i$$
$$s.t.\ \ y_i(\vec{w}\cdot\vec{x}_i - b) \geq 1 - \xi_i, \forall \vec{x}_i, \xi_i \geq 0$$

If you're still confused, don't worry, you're not alone. Luckily, SVM packages will happily optimize this for you without you having to understand the technical details. The important piece to understand is the addition of the cost parameter *C*. Modifying this value will adjust the penalty, for example, the fall on the wrong side of the hyperplane. The greater the cost parameter, the harder the optimization will try to achieve 100 percent separation. On the other hand, a lower cost parameter will place the emphasis on a wider overall margin. It is important to strike a balance between these two in order to create a model that generalizes well to future data.

Using kernels for non-linear spaces

In many real-world applications, the relationships between variables are nonlinear. As we just discovered, a SVM can still be trained on such data through the addition of a slack variable, which allows some examples to be misclassified. However, this is not the only way to approach the problem of nonlinearity. A key feature of SVMs is their ability to map the problem into a higher dimension space using a process known as the **kernel trick**. In doing so, a nonlinear relationship may suddenly appear to be quite linear.

Though this seems like nonsense, it is actually quite easy to illustrate by example. In the following figure, the scatterplot on the left depicts a nonlinear relationship between a weather class (sunny or snowy) and two features: latitude and longitude. The points at the center of the plot are members of the snowy class, while the points at the margins are all sunny. Such data could have been generated from a set of weather reports, some of which were obtained from stations near the top of a mountain, while others were obtained from stations around the base of the mountain.



On the right side of the figure, after the kernel trick has been applied, we look at the data through the lens of a new dimension: altitude. With the addition of this feature, the classes are now perfectly linearly separable. This is possible because we have obtained a new perspective on the data. In the left figure, we are viewing the mountain from a bird's eye view, while in the right one, we are viewing the mountain from a distance at the ground level. Here, the trend is obvious: snowy weather is found at higher altitudes.

SVMs with nonlinear kernels add additional dimensions to the data in order to create separation in this way. Essentially, the kernel trick involves a process of constructing new features that express mathematical relationships between measured characteristics. For instance, the altitude feature can be expressed mathematically as an interaction between latitude and longitude—the closer the point is to the center of each of these scales, the greater the altitude. This allows SVM to learn concepts that were not explicitly measured in the original data.

SVMs with nonlinear kernels are extremely powerful classifiers, although they do have some downsides, as shown in the following table:

Strengths	Weaknesses
• Can be used for classification or numeric prediction problems • Not overly influenced by noisy data and not very prone to overfitting • May be easier to use than neural networks, particularly due to the existence of several well-supported SVM algorithms • Gaining popularity due to its high accuracy and high-profile wins in data mining competitions	• Finding the best model requires testing of various combinations of kernels and model parameters • Can be slow to train, particularly if the input dataset has a large number of features or examples • Results in a complex black box model that is difficult, if not impossible, to interpret

Kernel functions, in general, are of the following form. The function denoted by the Greek letter phi, that is, $\varphi(x)$, is a mapping of the data into another space. Therefore, the general kernel function applies some transformation to the feature vectors x_i and x_j, and combines them using the **dot product**, which takes two vectors and returns a single number.

$$K(\vec{x_i}, \vec{x_j}) = \phi(\vec{x_i}) \cdot \phi(\vec{x_j})$$

Using this form, kernel functions have been developed for many different domains of data. A few of the most commonly used kernel functions are listed as follows. Nearly all SVM software packages will include these kernels, among many others.

The **linear kernel** does not transform the data at all. Therefore, it can be expressed simply as the dot product of the features:

$$K(\vec{x_i}, \vec{x_j}) = \vec{x_i} \cdot \vec{x_j}$$

The **polynomial kernel** of degree d adds a simple nonlinear transformation of the data:

$$K(\vec{x_i}, \vec{x_j}) = (\vec{x_i} \cdot \vec{x_j} + 1)^d$$

The **sigmoid kernel** results in a SVM model somewhat analogous to a neural network using a sigmoid activation function. The Greek letters kappa and delta are used as kernel parameters:

$$K(\vec{x_i}, \vec{x_j}) = \tanh(\kappa \, \vec{x_i} \cdot \vec{x_j} - \delta)$$

The **Gaussian RBF kernel** is similar to a RBF neural network. The RBF kernel performs well on many types of data and is thought to be a reasonable starting point for many learning tasks:

$$K(\vec{x_i}, \vec{x_j}) = e^{\frac{-\|\vec{x_i} - \vec{x_j}\|^2}{2\sigma^2}}$$

There is no reliable rule to match a kernel to a particular learning task. The fit depends heavily on the concept to be learned as well as the amount of training data and the relationships among the features. Often, a bit of trial and error is required by training and evaluating several SVMs on a validation dataset. This said, in many cases, the choice of kernel is arbitrary, as the performance may vary slightly. To see how this works in practice, let's apply our understanding of SVM classification to a real-world problem.

Example – performing OCR with SVMs

Image processing is a difficult task for many types of machine learning algorithms. The relationships linking patterns of pixels to higher concepts are extremely complex and hard to define. For instance, it's easy for a human being to recognize a face, a cat, or the letter "A", but defining these patterns in strict rules is difficult. Furthermore, image data is often noisy. There can be many slight variations in how the image was captured, depending on the lighting, orientation, and positioning of the subject.

SVMs are well-suited to tackle the challenges of image data. Capable of learning complex patterns without being overly sensitive to noise, they are able to recognize visual patterns with a high degree of accuracy. Moreover, the key weakness of SVMs—the black box model representation—is less critical for image processing. If an SVM can differentiate a cat from a dog, it does not matter much how it is doing so.

In this section, we will develop a model similar to those used at the core of the **Optical Character Recognition** (**OCR**) software often bundled with desktop document scanners. The purpose of such software is to process paper-based documents by converting printed or handwritten text into an electronic form to be saved in a database. Of course, this is a difficult problem due to the many variants in handwritten style and printed fonts. Even so, software users expect perfection, as errors or typos can result in embarrassing or costly mistakes in a business environment. Let's see whether our SVM is up to the task.

Step 1 – collecting data

When OCR software first processes a document, it divides the paper into a matrix such that each cell in the grid contains a single **glyph**, which is just a term referring to a letter, symbol, or number. Next, for each cell, the software will attempt to match the glyph to a set of all characters it recognizes. Finally, the individual characters would be combined back together into words, which optionally could be spell-checked against a dictionary in the document's language.

In this exercise, we'll assume that we have already developed the algorithm to partition the document into rectangular regions each consisting of a single character. We will also assume the document contains only alphabetic characters in English. Therefore, we'll simulate a process that involves matching glyphs to one of the 26 letters, A through Z.

To this end, we'll use a dataset donated to the UCI Machine Learning Data Repository (`http://archive.ics.uci.edu/ml`) by W. Frey and D. J. Slate. The dataset contains 20,000 examples of 26 English alphabet capital letters as printed using 20 different randomly reshaped and distorted black and white fonts.

For more information on this dataset, refer to Slate DJ, Frey W. Letter recognition using Holland-style adaptive classifiers. *Machine Learning*. 1991; 6:161-182.

The following figure, published by Frey and Slate, provides an example of some of the printed glyphs. Distorted in this way, the letters are challenging for a computer to identify, yet are easily recognized by a human being:

Step 2 – exploring and preparing the data

According to the documentation provided by Frey and Slate, when the glyphs are scanned into the computer, they are converted into pixels and 16 statistical attributes are recorded.

The attributes measure such characteristics as the horizontal and vertical dimensions of the glyph, the proportion of black (versus white) pixels, and the average horizontal and vertical position of the pixels. Presumably, differences in the concentration of black pixels across various areas of the box should provide a way to differentiate among the 26 letters of the alphabet.

To follow along with this example, download the `letterdata.csv` file from the Packt Publishing website, and save it to your R working directory.

Reading the data into R, we confirm that we have received the data with the 16 features that define each example of the letter class. As expected, letter has 26 levels:

```
> letters <- read.csv("letterdata.csv")
> str(letters)
'data.frame':  20000 obs. of 17 variables:
 $ letter: Factor w/ 26 levels "A","B","C","D",..
 $ xbox  : int  2 5 4 7 2 4 4 1 2 11 ...
 $ ybox  : int  8 12 11 11 1 11 2 1 2 15 ...
 $ width : int  3 3 6 6 3 5 5 3 4 13 ...
 $ height: int  5 7 8 6 1 8 4 2 4 9 ...
```

```
$ onpix : int  1 2 6 3 1 3 4 1 2 7 ...
$ xbar  : int  8 10 10 5 8 8 8 8 10 13 ...
$ ybar  : int  13 5 6 9 6 8 7 2 6 2 ...
$ x2bar : int  0 5 2 4 6 6 6 2 2 6 ...
$ y2bar : int  6 4 6 6 6 9 6 2 6 2 ...
$ xybar : int  6 13 10 4 6 5 7 8 12 12 ...
$ x2ybar: int  10 3 3 4 5 6 6 2 4 1 ...
$ xy2bar: int  8 9 7 10 9 6 6 8 8 9 ...
$ xedge : int  0 2 3 6 1 0 2 1 1 8 ...
$ xedgey: int  8 8 7 10 7 8 8 6 6 1 ...
$ yedge : int  0 4 3 2 5 9 7 2 1 1 ...
$ yedgex: int  8 10 9 8 10 7 10 7 7 8 ...
```

Recall that SVM learners require all features to be numeric, and moreover, that each feature is scaled to a fairly small interval. In this case, every feature is an integer, so we do not need to convert any factors into numbers. On the other hand, some of the ranges for these integer variables appear fairly wide. This indicates that we need to normalize or standardize the data. However, we can skip this step for now, because the R package that we will use for fitting the SVM model will perform the rescaling automatically.

Given that the data preparation has been largely done for us, we can move directly to the training and testing phases of the machine learning process. In the previous analyses, we randomly divided the data between the training and testing sets. Although we could do so here, Frey and Slate have already randomized the data, and therefore suggest using the first 16,000 records (80 percent) to build the model and the next 4,000 records (20 percent) to test. Following their advice, we can create training and testing data frames as follows:

```
> letters_train <- letters[1:16000, ]
> letters_test  <- letters[16001:20000, ]
```

With our data ready to go, let's start building our classifier.

Step 3 – training a model on the data

When it comes to fitting an SVM model in R, there are several outstanding packages to choose from. The `e1071` package from the Department of Statistics at the Vienna University of Technology (TU Wien) provides an R interface to the award winning LIBSVM library, a widely used open source SVM program written in C++. If you are already familiar with LIBSVM, you may want to start here.

For more information on LIBSVM, refer to the authors' website at `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`.

Similarly, if you're already invested in the SVMlight algorithm, the `klaR` package from the Department of Statistics at the Dortmund University of Technology (TU Dortmund) provides functions to work with this SVM implementation directly within R.

For information on SVMlight, have a look at `http://svmlight.joachims.org/`.

Finally, if you are starting from scratch, it is perhaps best to begin with the SVM functions in the `kernlab` package. An interesting advantage of this package is that it was developed natively in R rather than C or C++, which allows it to be easily customized; none of the internals are hidden behind the scenes. Perhaps even more importantly, unlike the other options, `kernlab` can be used with the `caret` package, which allows SVM models to be trained and evaluated using a variety of automated methods (covered in *Chapter 11, Improving Model Performance*).

For a more thorough introduction to `kernlab`, please refer to the authors' paper at `http://www.jstatsoft.org/v11/i09/`.

The syntax for training SVM classifiers with `kernlab` is as follows. If you do happen to be using one of the other packages, the commands are largely similar. By default, the `ksvm()` function uses the Gaussian RBF kernel, but a number of other options are provided.

Support vector machine syntax

using the `ksvm()` function in the `kernlab` package

Building the model:

```
m <- ksvm(target ~ predictors, data = mydata,
          kernel = "rbfdot", C = 1)
```

- `target` is the outcome in the `mydata` data frame to be modeled
- `predictors` is an R formula specifying the features in the `mydata` data frame to use for prediction
- `data` specifies the data frame in which the `target` and `predictors` variables can be found
- `kernel` specifies a nonlinear mapping such as `"rbfdot"` (radial basis), `"polydot"` (polynomial), `"tanhdot"` (hyperbolic tangent sigmoid), or `"vanilladot"` (linear)
- `C` is a number that specifies the cost of violating the constraints, i.e., how big of a penalty there is for the "soft margin." Larger values will result in narrower margins

The function will return a SVM object that can be used to make predictions.

Making predictions:

```
p <- predict(m, test, type = "response")
```

- `m` is a model trained by the `ksvm()` function
- `test` is a data frame containing test data with the same features as the training data used to build the classifier
- `type` specifies whether the predictions should be `"response"` (the predicted class) or `"probabilities"` (the predicted probability, one column per class level).

The function will return a vector (or matrix) of predicted classes (or probabilities) depending on the value of the type parameter.

Example:

```
letter_classifier <- ksvm(letter ~ ., data =
  letters_train, kernel = "vanilladot")
letter_prediction <- predict(letter_classifier,
  letters_test)
```

To provide a baseline measure of SVM performance, let's begin by training a simple linear SVM classifier. If you haven't already, install the `kernlab` package to your library, using the `install.packages("kernlab")` command. Then, we can call the `ksvm()` function on the training data and specify the linear (that is, vanilla) kernel using the `vanilladot` option, as follows:

```
> library(kernlab)
> letter_classifier <- ksvm(letter ~ ., data = letters_train,
                            kernel = "vanilladot")
```

Depending on the performance of your computer, this operation may take some time to complete. When it finishes, type the name of the stored model to see some basic information about the training parameters and the fit of the model.

```
> letter_classifier
Support Vector Machine object of class "ksvm"

SV type: C-svc  (classification)
 parameter : cost C = 1

Linear (vanilla) kernel function.

Number of Support Vectors : 7037

Objective Function Value : -14.1746 -20.0072 -23.5628 -6.2009 -7.5524
-32.7694 -49.9786 -18.1824 -62.1111 -32.7284 -16.2209...

Training error : 0.130062
```

This information tells us very little about how well the model will perform in the real world. We'll need to examine its performance on the testing dataset to know whether it generalizes well to unseen data.

Step 4 – evaluating model performance

The `predict()` function allows us to use the letter classification model to make predictions on the testing dataset:

```
> letter_predictions <- predict(letter_classifier, letters_test)
```

Because we didn't specify the type parameter, the `type = "response"` default was used. This returns a vector containing a predicted letter for each row of values in the test data. Using the `head()` function, we can see that the first six predicted letters were `U`, `N`, `V`, `X`, `N`, and `H`:

```
> head(letter_predictions)
[1] U N V X N H
Levels: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

To examine how well our classifier performed, we need to compare the predicted letter to the true letter in the testing dataset. We'll use the `table()` function for this purpose (only a portion of the full table is shown here):

```
> table(letter_predictions, letters_test$letter)

letter_predictions   A   B   C   D   E
                 A 144   0   0   0   0
                 B   0 121   0   5   2
                 C   0   0 120   0   4
                 D   2   2   0 156   0
                 E   0   0   5   0 127
```

The diagonal values of `144`, `121`, `120`, `156`, and `127` indicate the total number of records where the predicted letter matches the true value. Similarly, the number of mistakes is also listed. For example, the value of `5` in row `B` and column `D` indicates that there were five cases where the letter `D` was misidentified as a `B`.

Looking at each type of mistake individually may reveal some interesting patterns about the specific types of letters the model has trouble with, but this is time consuming. We can simplify our evaluation instead by calculating the overall accuracy. This considers only whether the prediction was correct or incorrect, and ignores the type of error.

The following command returns a vector of `TRUE` or `FALSE` values, indicating whether the model's predicted letter agrees with (that is, matches) the actual letter in the test dataset:

```
> agreement <- letter_predictions == letters_test$letter
```

Using the `table()` function, we see that the classifier correctly identified the letter in 3,357 out of the 4,000 test records:

```
> table(agreement)

agreement
FALSE  TRUE
  643  3357
```

In percentage terms, the accuracy is about 84 percent:

```
> prop.table(table(agreement))

agreement
  FALSE    TRUE
0.16075 0.83925
```

Note that when Frey and Slate published the dataset in 1991, they reported a recognition accuracy of about 80 percent. Using just a few lines of R code, we were able to surpass their result, although we also have the benefit of over two decades of additional machine learning research. With this in mind, it is likely that we are able to do even better.

Step 5 – improving model performance

Our previous SVM model used the simple linear kernel function. By using a more complex kernel function, we can map the data into a higher dimensional space, and potentially obtain a better model fit.

It can be challenging, however, to choose from the many different kernel functions. A popular convention is to begin with the Gaussian RBF kernel, which has been shown to perform well for many types of data. We can train an RBF-based SVM, using the `ksvm()` function as shown here:

```
> letter_classifier_rbf <- ksvm(letter ~ ., data = letters_train,
                                kernel = "rbfdot")
```

Next, we make predictions as done earlier:

```
> letter_predictions_rbf <- predict(letter_classifier_rbf,
                                    letters_test)
```

Finally, we'll compare the accuracy to our linear SVM:

```
> agreement_rbf <- letter_predictions_rbf == letters_test$letter
> table(agreement_rbf)
agreement_rbf
FALSE  TRUE
  275  3725
> prop.table(table(agreement_rbf))
agreement_rbf
  FALSE    TRUE
0.06875 0.93125
```

Your results may differ from those shown here due to randomness in the `ksvm` RBF kernel. If you'd like them to match exactly, use `set.seed(12345)` prior to running the ksvm() function.

By simply changing the kernel function, we were able to increase the accuracy of our character recognition model from 84 percent to 93 percent. If this level of performance is still unsatisfactory for the OCR program, other kernels could be tested, or the cost of constraints parameter C could be varied to modify the width of the decision boundary. As an exercise, you should experiment with these parameters to see how they impact the success of the final model.

Summary

In this chapter, we examined two machine learning methods that offer a great deal of potential, but are often overlooked due to their complexity. Hopefully, you now see that this reputation is at least somewhat undeserved. The basic concepts that drive ANNs and SVMs are fairly easy to understand.

On the other hand, because ANNs and SVMs have been around for many decades, each of them has numerous variations. This chapter just scratches the surface of what is possible with these methods. By utilizing the terminology you learned here, you should be capable of picking up the nuances that distinguish the many advancements that are being developed every day.

Now that we have spent some time learning about many different types of predictive models from simple to sophisticated; in the next chapter, we will begin to consider methods for other types of learning tasks. These unsupervised learning techniques will bring to light fascinating patterns within the data.

8

Finding Patterns – Market Basket Analysis Using Association Rules

Think back to the last time you made an impulse purchase. Maybe you were waiting in the grocery store checkout lane and bought a pack of chewing gum or a candy bar. Perhaps on a late-night trip for diapers and formula you picked up a caffeinated beverage or a six-pack of beer. You might have even bought this book on a whim on a bookseller's recommendation. These impulse buys are no coincidence, as retailers use sophisticated data analysis techniques to identify patterns that will drive retail behavior.

In years past, such recommendation systems were based on the subjective intuition of marketing professionals and inventory managers or buyers. More recently, as barcode scanners, computerized inventory systems, and online shopping trends have built a wealth of transactional data, machine learning has been increasingly applied to learn purchasing patterns. The practice is commonly known as **market basket analysis** due to the fact that it has been so frequently applied to supermarket data.

Although the technique originated with shopping data, it is useful in other contexts as well. By the time you finish this chapter, you will be able to apply market basket analysis techniques to your own tasks, whatever they may be. Generally, the work involves:

- Using simple performance measures to find associations in large databases
- Understanding the peculiarities of transactional data
- Knowing how to identify the useful and actionable patterns

The results of a market basket analysis are actionable patterns. Thus, as we apply the technique, you are likely to identify applications to your work, even if you have no affiliation with a retail chain.

Understanding association rules

The building blocks of a market basket analysis are the items that may appear in any given transaction. Groups of one or more items are surrounded by brackets to indicate that they form a set, or more specifically, an **itemset** that appears in the data with some regularity. Transactions are specified in terms of itemsets, such as the following transaction that might be found in a typical grocery store:

{bread, peanut butter, jelly}

The result of a market basket analysis is a collection of **association rules** that specify patterns found in the relationships among items he itemsets. Association rules are always composed from subsets of itemsets and are denoted by relating one itemset on the left-hand side (LHS) of the rule to another itemset on the right-hand side (RHS) of the rule. The LHS is the condition that needs to be met in order to trigger the rule, and the RHS is the expected result of meeting that condition. A rule identified from the example transaction might be expressed in the form:

{peanut butter, jelly} → {bread}

In plain language, this association rule states that if peanut butter and jelly are purchased together, then bread is also likely to be purchased. In other words, "peanut butter and jelly imply bread."

Developed in the context of retail transaction databases, association rules are not used for prediction, but rather for unsupervised knowledge discovery in large databases. This is unlike the classification and numeric prediction algorithms presented in previous chapters. Even so, you will find that association rule learners are closely related to and share many features of the classification rule learners presented in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules.*

Because association rule learners are unsupervised, there is no need for the algorithm to be trained; data does not need to be labeled ahead of time. The program is simply unleashed on a dataset in the hope that interesting associations are found. The downside, of course, is that there isn't an easy way to objectively measure the performance of a rule learner, aside from evaluating them for qualitative usefulness—typically, an eyeball test of some sort.

Although association rules are most often used for market basket analysis, they are helpful for finding patterns in many different types of data. Other potential applications include:

- Searching for interesting and frequently occurring patterns of DNA and protein sequences in cancer data
- Finding patterns of purchases or medical claims that occur in combination with fraudulent credit card or insurance use
- Identifying combinations of behavior that precede customers dropping their cellular phone service or upgrading their cable television package

Association rule analysis is used to search for interesting connections among a very large number of elements. Human beings are capable of such insight quite intuitively, but it often takes expert-level knowledge or a great deal of experience to do what a rule learning algorithm can do in minutes or even seconds. Additionally, some datasets are simply too large and complex for a human being to find the needle in the haystack.

The Apriori algorithm for association rule learning

Just as it is challenging for humans, transactional data makes association rule mining a challenging task for machines as well. Transactional datasets are typically extremely large, both in terms of the number of transactions as well as the number of items or features that are monitored. The problem is that the number of potential itemsets grows exponentially with the number of features. Given *k* items that can appear or not appear in a set, there are *2^k* possible itemsets that could be potential rules. A retailer that sells only 100 different items could have on the order of *2^100 = 1.27e+30* itemsets that an algorithm must evaluate—a seemingly impossible task.

Rather than evaluating each of these itemsets one by one, a smarter rule learning algorithm takes advantage of the fact that, in reality, many of the potential combinations of items are rarely, if ever, found in practice. For instance, even if a store sells both automotive items and women's cosmetics, a set of *{motor oil, lipstick}* is likely to be extraordinarily uncommon. By ignoring these rare (and, perhaps, less important) combinations, it is possible to limit the scope of the search for rules to a more manageable size.

Much work has been done to identify heuristic algorithms for reducing the number of itemsets to search. Perhaps the most-widely used approach for efficiently searching large databases for rules is known as **Apriori**. Introduced in 1994 by Rakesh Agrawal and Ramakrishnan Srikant, the Apriori algorithm has since become somewhat synonymous with association rule learning. The name is derived from the fact that the algorithm utilizes a simple prior (that is, *a priori*) belief about the properties of frequent itemsets.

Before we discuss that in more depth, it's worth noting that this algorithm, like all learning algorithms, is not without its strengths and weaknesses. Some of these are listed as follows:

Strengths	Weaknesses
• Is capable of working with large amounts of transactional data • Results in rules that are easy to understand • Useful for "data mining" and discovering unexpected knowledge in databases	• Not very helpful for small datasets • Requires effort to separate the true insight from common sense • Easy to draw spurious conclusions from random patterns

As noted earlier, the Apriori algorithm employs a simple *a priori* belief to reduce the association rule search space: all subsets of a frequent itemset must also be frequent. This heuristic is known as the **Apriori property**. Using this astute observation, it is possible to dramatically limit the number of rules to be searched. For example, the set *{motor oil, lipstick}* can only be frequent if both *{motor oil}* and *{lipstick}* occur frequently as well. Consequently, if either motor oil or lipstick is infrequent, any set containing these items can be excluded from the search.

For additional details on the Apriori algorithm, refer to: Agrawal R, Srikant R. Fast algorithms for mining association rules. *Proceedings of the 20th International Conference on Very Large Databases*. 1994:487-499.

To see how this principle can be applied in a more realistic setting, let's consider a simple transaction database. The following table shows five completed transactions in an imaginary hospital's gift shop:

Transaction number	Purchased items
1	*{flowers, get well card, soda}*
2	*{plush toy bear, flowers, balloons, candy bar}*
3	*{get well card, candy bar, flowers}*
4	*{plush toy bear, balloons, soda}*
5	*{flowers, get well card, soda}*

By looking at the sets of purchases, one can infer that there are a couple of typical buying patterns. A person visiting a sick friend or family member tends to buy a get well card and flowers, while visitors to new mothers tend to buy plush toy bears and balloons. Such patterns are notable because they appear frequently enough to catch our interest; we simply apply a bit of logic and subject matter experience to explain the rule.

In a similar fashion, the Apriori algorithm uses statistical measures of an itemset's "interestingness" to locate association rules in much larger transaction databases. In the sections that follow, we will discover how Apriori computes such measures of interest and how they are combined with the Apriori property to reduce the number of rules to be learned.

Measuring rule interest – support and confidence

Whether or not an association rule is deemed interesting is determined by two statistical measures: support and confidence measures. By providing minimum thresholds for each of these metrics and applying the Apriori principle, it is easy to drastically limit the number of rules reported, perhaps even to the point where only the obvious or common sense rules are identified. For this reason, it is important to carefully understand the types of rules that are excluded under these criteria.

The **support** of an itemset or rule measures how frequently it occurs in the data. For instance the itemset *{get well card, flowers}*, has support of *3 / 5 = 0.6* in the hospital gift shop data. Similarly, the support for *{get well card}* → *{flowers}* is also 0.6. The support can be calculated for any itemset or even a single item; for instance, the support for *{candy bar}* is *2 / 5 = 0.4*, since candy bars appear in 40 percent of purchases. A function defining support for the itemset *X* can be defined as follows:

$$\text{support}(X) = \frac{\text{count}(X)}{N}$$

Here, *N* is the number of transactions in the database and *count(X)* is the number of transactions containing itemset *X*.

A rule's **confidence** is a measurement of its predictive power or accuracy. It is defined as the support of the itemset containing both *X* and *Y* divided by the support of the itemset containing only *X*:

$$\text{confidence}(X \rightarrow Y) = \frac{\text{support}(X, Y)}{\text{support}(X)}$$

Essentially, the confidence tells us the proportion of transactions where the presence of item or itemset *X* results in the presence of item or itemset *Y*. Keep in mind that the confidence that *X* leads to *Y* is not the same as the confidence that *Y* leads to *X*. For example, the confidence of *{flowers}* → *{get well card}* is *0.6 / 0.8 = 0.75*. In comparison, the confidence of *{get well card}* → *{flowers}* is *0.6 / 0.6 = 1.0*. This means that a purchase involving flowers is accompanied by a purchase of a get well card 75 percent of the time, while a purchase of a get well card is associated with flowers 100 percent of the time. This information could be quite useful to the gift shop management.

You may have noticed similarities between support, confidence, and the Bayesian probability rules covered in *Chapter 4, Probabilistic Learning – Classification Using Naive Bayes*. In fact, *support(A, B)* is the same as *P(A∩B)* and *confidence(A → B)* is the same as *P(B | A)*. It is just the context that differs.

Rules like *{get well card}* → *{flowers}* are known as **strong rules**, because they have both high support and confidence. One way to find more strong rules would be to examine every possible combination of the items in the gift shop, measure the support and confidence value, and report back only those rules that meet certain levels of interest. However, as noted before, this strategy is generally not feasible for anything but the smallest of datasets.

In the next section, you will see how the Apriori algorithm uses the minimum levels of support and confidence with the Apriori principle to find strong rules quickly by reducing the number of rules to a more manageable level.

Building a set of rules with the Apriori principle

Recall that the Apriori principle states that all subsets of a frequent itemset must also be frequent. In other words, if *{A, B}* is frequent, then *{A}* and *{B}* must both be frequent. Recall also that by definition, the support indicates how frequently an itemset appears in the data. Therefore, if we know that *{A}* does not meet a desired support threshold, there is no reason to consider *{A, B}* or any itemset containing *{A}*; it cannot possibly be frequent.

The Apriori algorithm uses this logic to exclude potential association rules prior to actually evaluating them. The actual process of creating rules occurs in two phases:

1. Identifying all the itemsets that meet a minimum support threshold.
2. Creating rules from these itemsets using those meeting a minimum confidence threshold.

The first phase occurs in multiple iterations. Each successive iteration involves evaluating the support of a set of increasingly large itemsets. For instance, iteration 1 involves evaluating the set of 1-item itemsets (1-itemsets), iteration 2 evaluates 2-itemsets, and so on. The result of each iteration *i* is a set of all the *i*-itemsets that meet the minimum support threshold.

All the itemsets from iteration *i* are combined in order to generate candidate itemsets for the evaluation in iteration *i + 1*. But the Apriori principle can eliminate some of them even before the next round begins. If *{A}*, *{B}*, and *{C}* are frequent in iteration 1 while *{D}* is not frequent, iteration 2 will consider only *{A, B}*, *{A, C}*, and *{B, C}*. Thus, the algorithm needs to evaluate only three itemsets rather than the six that would have been evaluated if the sets containing *D* had not been eliminated *a priori*.

Continuing with this thought, suppose during iteration 2, it is discovered that *{A, B}* and *{B, C}* are frequent, but *{A, C}* is not. Although iteration 3 would normally begin by evaluating the support for *{A, B, C}*, it is not mandatory that this step should occur at all. Why not? The Apriori principle states that *{A, B, C}* cannot possibly be frequent, since the subset *{A, C}* is not. Therefore, having generated no new itemsets in iteration 3, the algorithm may stop.

At this point, the second phase of the Apriori algorithm may begin. Given the set of frequent itemsets, association rules are generated from all possible subsets. For instance, *{A, B}* would result in candidate rules for *{A} → {B}* and *{B} → {A}*. These are evaluated against a minimum confidence threshold, and any rule that does not meet the desired confidence level is eliminated.

Example – identifying frequently purchased groceries with association rules

As noted in this chapter's introduction, market basket analysis is used behind the scenes for the recommendation systems used in many brick-and-mortar and online retailers. The learned association rules indicate the combinations of items that are often purchased together. Knowledge of these patterns provides insight into new ways a grocery chain might optimize the inventory, advertise promotions, or organize the physical layout of the store. For instance, if shoppers frequently purchase coffee or orange juice with a breakfast pastry, it may be possible to increase profit by relocating pastries closer to coffee and juice.

In this tutorial, we will perform a market basket analysis of transactional data from a grocery store. However, the techniques could be applied to many different types of problems, from movie recommendations, to dating sites, to finding dangerous interactions among medications. In doing so, we will see how the Apriori algorithm is able to efficiently evaluate a potentially massive set of association rules.

Step 1 – collecting data

Our market basket analysis will utilize the purchase data collected from one month of operation at a real-world grocery store. The data contains 9,835 transactions or about 327 transactions per day (roughly 30 transactions per hour in a 12-hour business day), suggesting that the retailer is not particularly large, nor is it particularly small.

The dataset used here was adapted from the `Groceries` dataset in the `arules` R package. For more information, see: Hahsler M, Hornik K, Reutterer T. Implications of probabilistic data modeling for mining association rules. In: Gaul W, Vichi M, Weihs C, ed. *Studies in Classification, Data Analysis, and Knowledge Organization: from Data and Information Analysis to Knowledge Engineering*. New York: Springer; 2006:598–605.

The typical grocery store offers a huge variety of items. There might be five brands of milk, a dozen different types of laundry detergent, and three brands of coffee. Given the moderate size of the retailer, we will assume that they are not terribly concerned with finding rules that apply only to a specific brand of milk or detergent. With this in mind, all brand names can be removed from the purchases. This reduces the number of groceries to a more manageable 169 types, using broad categories such as chicken, frozen meals, margarine, and soda.

If you hope to identify highly specific association rules—such as whether customers prefer grape or strawberry jelly with their peanut butter—you will need a tremendous amount of transactional data. Large chain retailers use databases of many millions of transactions in order to find associations among particular brands, colors, or flavors of items.

Do you have any guesses about which types of items might be purchased together? Will wine and cheese be a common pairing? Bread and butter? Tea and honey? Let's dig into this data and see whether we can confirm our guesses.

Step 2 – exploring and preparing the data

Transactional data is stored in a slightly different format than that we used previously. Most of our prior analyses utilized data in the matrix form where rows indicated example instances and columns indicated features. Given the structure of the matrix format, all examples are required to have exactly the same set of features.

In comparison, transactional data is a more free form. As usual, each row in the data specifies a single example—in this case, a transaction. However, rather than having a set number of features, each record comprises a comma-separated list of any number of items, from one to many. In essence, the features may differ from example to example.

To follow along with this analysis, download the `groceries.csv` file from the Packt Publishing website and save it in your R working directory.

The first five rows of the raw `grocery.csv` file are as follows:

```
citrus fruit,semi-finished bread,margarine,ready soups
tropical fruit,yogurt,coffee
whole milk
pip fruit,yogurt,cream cheese,meat spreads
other vegetables,whole milk,condensed milk,long life bakery
product
```

These lines indicate five separate grocery store transactions. The first transaction included four items: citrus fruit, semi-finished bread, margarine, and ready soups. In comparison, the third transaction included only one item: whole milk.

Suppose we try to load the data using the `read.csv()` function as we did in the prior analyses. R would happily comply and read the data into a matrix form as follows:

	V1	V2	V3	V4
1	citrus fruit	semi-finished bread	margarine	ready soups
2	tropical fruit	yogurt	coffee	
3	whole milk			
4	pip fruit	yogurt	cream cheese	meat spreads
5	other vegetables	whole milk	condensed milk	long life bakery product

You will notice that R created four columns to store the items in the transactional data: `V1`, `V2`, `V3`, and `V4`. Although this may seem reasonable this, if we use the data in this form, we will encounter problems later may seem reasonable, R chose to create four variables because the first line had exactly four comma-separated values. However, we know that grocery purchases can contain more than four items; in the four column design such transactions will be broken across multiple rows in the matrix. We could try to remedy this by putting the transaction with the largest number of items at the top of the file, but this ignores another more problematic issue.

By structuring data this way, R has constructed a set of features that record not just the items in the transactions, but also the order in which they appear. If we imagine our learning algorithm as an attempt to find a relationship among `V1`, `V2`, `V3`, and `V4`, then whole milk in `V1` might be treated differently than the whole milk appearing in `V2`. Instead, we need a dataset that does not treat a transaction as a set of positions to be filled (or not filled) with specific items, but rather as a market basket that either contains or does not contain each particular item.

Data preparation – creating a sparse matrix for transaction data

The solution to this problem utilizes a data structure called a **sparse matrix**. You may recall that we used a sparse matrix to process text data in *Chapter 4, Probabilistic Learning – Classification Using Naive Bayes*. Just as with the preceding dataset, each row in the sparse matrix indicates a transaction. However, the sparse matrix has a column (that is, feature) for every item that could possibly appear in someone's shopping bag. Since there are 169 different items in our grocery store data, our sparse matrix will contain 169 columns.

Why not just store this as a data frame as we did in most of our analyses? The reason is that as additional transactions and items are added, a conventional data structure quickly becomes too large to fit in the available memory. Even with the relatively small transactional dataset used here, the matrix contains nearly 1.7 million cells, most of which contain zeros (hence, the name "sparse" matrix—there are very few nonzero values). Since there is no benefit to storing all these zero values, a sparse matrix does not actually store the full matrix in memory; it only stores the cells that are occupied by an item. This allows the structure to be more memory efficient than an equivalently sized matrix or data frame.

In order to create the sparse matrix data structure from the transactional data, we can use the functionality provided by the `arules` package. Install and load the package using the `install.packages("arules")` and `library(arules)` commands.

For more information on the arules package, refer to: Hahsler M, Gruen B, Hornik K. arules – a computational environment for mining association rules and frequent item sets. *Journal of Statistical Software. 2005; 14.*

Since we're loading the transactional data, we cannot simply use the `read.csv()` function used previously. Instead, `arules` provides a `read.transactions()` function that is similar to `read.csv()` with the exception that it results in a sparse matrix suitable for transactional data. The `sep = ","` parameter specifies that items in the input file are separated by a comma. To read the `groceries.csv` data into a sparse matrix named `groceries`, type the following line:

```
> groceries <- read.transactions("groceries.csv", sep = ",")
```

To see some basic information about the `groceries` matrix we just created, use the `summary()` function on the object:

```
> summary(groceries)
transactions as itemMatrix in sparse format with
 9835 rows (elements/itemsets/transactions) and
 169 columns (items) and a density of 0.02609146
```

The first block of information in the output (as shown previously) provides a summary of the sparse matrix we created. The output `9835 rows` refers to the number of transactions, and the output `169 columns` refers to the 169 different items that might appear in someone's grocery basket. Each cell in the matrix is `1` if the item was purchased for the corresponding transaction, or `0` otherwise.

The **density** value of `0.02609146` (2.6 percent) refers to the proportion of nonzero matrix cells. Since there are *9,835 * 169 = 1,662,115* positions in the matrix, we can calculate that a total of *1,662,115 * 0.02609146 = 43,367* items were purchased during the store's 30 days of operation (ignoring the fact that duplicates of the same items might have been purchased). With an additional step, we can determine that the average transaction contained *43,367 / 8,835 = 4.409* distinct grocery items. Of course, if we look a little further down the output, we'll see that the mean number of items per transaction has already been provided.

The next block of the `summary()` output lists the items that were most commonly found in the transactional data. Since *2,513 / 9,835 = 0.2555*, we can determine that whole milk appeared in 25.6 percent of the transactions. The other vegetables, rolls/buns, soda, and yogurt round out the list of other common items, as follows:

```
most frequent items:
      whole milk other vegetables       rolls/buns
            2513             1903             1809
            soda           yogurt          (Other)
            1715             1372            34055
```

Finally, we are presented with a set of statistics about the size of the transactions. A total of 2,159 transactions contained only a single item, while one transaction had 32 items. The first quartile and median purchase sizes are two and three items, respectively, implying that 25 percent of the transactions contained two or fewer items and the transactions were split in half between those with less than three items and those with more. The mean of 4.409 items per transaction matches the value we calculated by hand.

```
element (itemset/transaction) length distribution:
sizes
   1    2    3    4    5    6    7    8    9   10   11   12
2159 1643 1299 1005  855  645  545  438  350  246  182  117
  13   14   15   16   17   18   19   20   21   22   23   24
  78   77   55   46   29   14   14    9   11    4    6    1
  26   27   28   29   32
   1    1    1    3    1

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.000   2.000   3.000   4.409   6.000  32.000
```

The `arules` package includes some useful features for examining transaction data. To look at the contents of the sparse matrix, use the `inspect()` function in combination with the vector operators. The first five transactions can be viewed as follows:

```
> inspect(groceries[1:5])
  items
1 {citrus fruit,
   margarine,
   ready soups,
   semi-finished bread}
2 {coffee,
   tropical fruit,
   yogurt}
3 {whole milk}
4 {cream cheese,
   meat spreads,
   pip fruit,
   yogurt}
5 {condensed milk,
   long life bakery product,
   other vegetables,
   whole milk}
```

These transactions match our look at the original CSV file. To examine a particular item (that is, a column of data), it is possible use the `[row, column]` matrix notion. Using this with the `itemFrequency()` function allows us to see the proportion of transactions that contain the item. This allows us, for instance, to view the support level for the first three items in the grocery data:

```
> itemFrequency(groceries[, 1:3])
abrasive cleaner artif. sweetener   baby cosmetics
    0.0035587189     0.0032536858     0.0006100661
```

Note that the items in the sparse matrix are sorted in columns by alphabetical order. Abrasive cleaner and artificial sweeteners are found in about 0.3 percent of the transactions, while baby cosmetics are found in about 0.06 percent of the transactions.

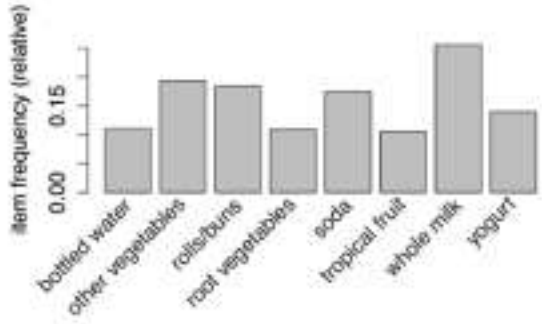
Visualizing item support – item frequency plots

To present these statistics visually, use the `itemFrequencyPlot()` function. This allows you to produce a bar chart depicting the proportion of transactions containing certain items. Since the transactional data contains a very large number of items, you will often need to limit the ones appearing in the plot in order to produce a legible chart.

If you require these items to appear in a minimum proportion of transactions, use `itemFrequencyPlot()` with the `support` parameter:

```
> itemFrequencyPlot(groceries, support = 0.1)
```

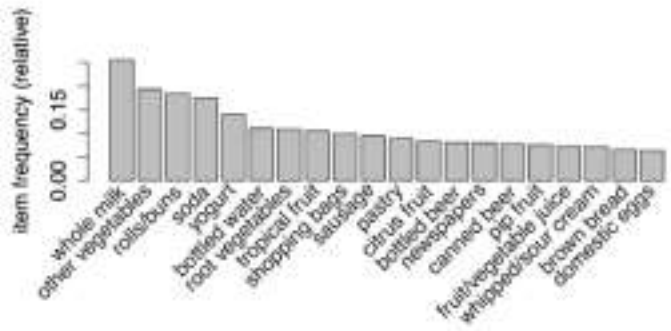
As shown in the following plot, this results in a histogram showing the eight items in the `groceries` data with at least 10 percent support:



If you would rather limit the plot to a specific number of items, the `topN` parameter can be used with `itemFrequencyPlot()`:

```
> itemFrequencyPlot(groceries, topN = 20)
```

The histogram is then sorted by decreasing support, as shown in the following diagram of the top 20 items in the `groceries` data:

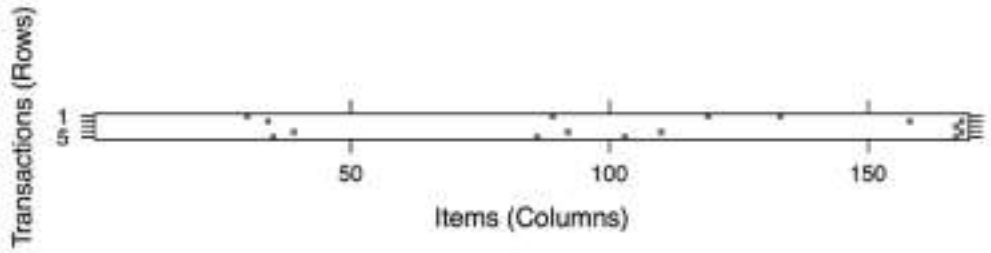


Visualizing the transaction data – plotting the sparse matrix

In addition to looking at the items, it's also possible to visualize the entire sparse matrix. To do so, use the `image()` function. The command to display the sparse matrix for the first five transactions is as follows:

```
> image(groceries[1:5])
```

The resulting diagram depicts a matrix with 5 rows and 169 columns, indicating the 5 transactions and 169 possible items we requested. Cells in the matrix are filled with black for transactions (rows) where the item (column) was purchased.



Although the preceding diagram is small and may be slightly hard to read, you can see that the first, fourth, and fifth transactions contained four items each, since their rows have four cells filled in. You can also see that rows three, five, two, and four have an item in common (on the right side of the diagram).

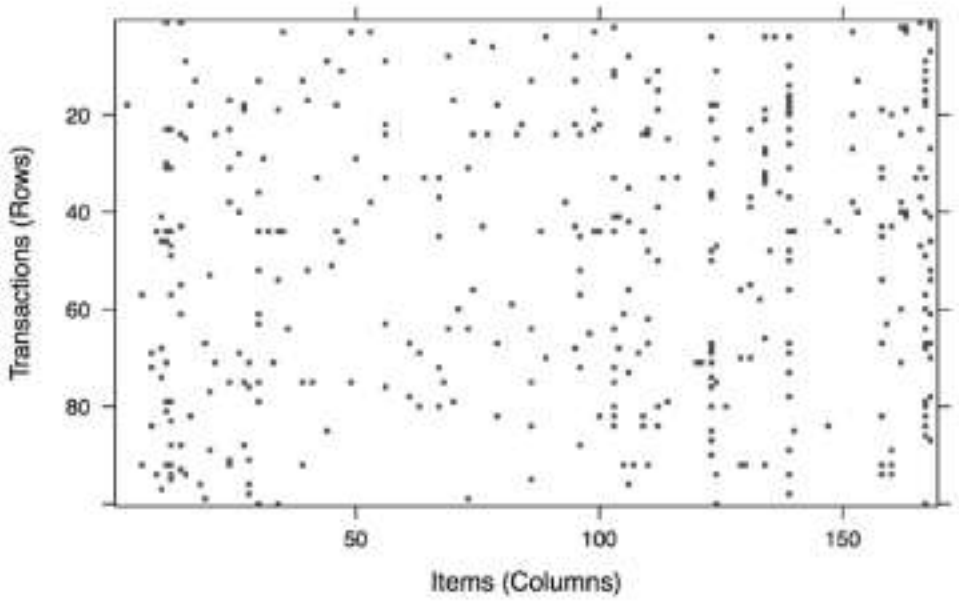
This visualization can be a useful tool for exploring data. For one, it may help with the identification of potential data issues. Columns that are filled all the way down could indicate items that are purchased in every transaction—a problem that could arise, perhaps, if a retailer's name or identification number was inadvertently included in the transaction dataset.

Additionally, patterns in the diagram may help reveal actionable insights within the transactions and items, particularly if the data is sorted in interesting ways. For example, if the transactions are sorted by date, the patterns in black dots could reveal seasonal effects in the number or types of items purchased. Perhaps around Christmas or Hanukkah, toys are more common; around Halloween, perhaps candies become popular. This type of visualization could be especially powerful if the items were also sorted into categories. In most cases, however, the plot will look fairly random, like static on a television screen.

Keep in mind that this visualization will not be as useful for extremely large transaction databases, because the cells will be too small to discern. Still, by combining it with the `sample()` function, you can view the sparse matrix for a randomly sampled set of transactions. The command to create random selection of 100 transactions is as follows:

```
> image(sample(groceries, 100))
```

This creates a matrix diagram with 100 rows and 169 columns:



A few columns seem fairly heavily populated, indicating some very popular items at the store. But overall, the distribution of dots seems fairly random. Given nothing else of note, let's continue with our analysis.

Step 3 – training a model on the data

With data preparation completed, we can now work at finding associations among shopping cart items. We will use an implementation of the Apriori algorithm in the `arules` package we've been using to explore and prepare the groceries data. You'll need to install and load this package if you have not done so already. The following table shows the syntax to create sets of rules with the `apriori()` function:

Association rule syntax

using the `apriori()` function in the `arules` package

Finding association rules:

```
myrules <- apriori(data = mydata, parameter =
  list(support = 0.1, confidence = 0.8, minlen = 1))
```

- `data` is a sparse item matrix holding transactional data
- `support` specifies the minimum required rule support
- `confidence` specifies the minimum required rule confidence
- `minlen` specifies the minimum required rule items

The function will return a rules object storing all rules that meet the minimum criteria.

Examining association rules:

```
inspect(myrules)
```

- `myrules` is a set of association rules from the `apriori()` function

This will output the association rules to the screen. Vector operators can be used on myrules to choose a specific rule or rules to view.

Example:

```
groceryrules <- apriori(groceries, parameter =
 list(support = 0.01, confidence = 0.25, minlen = 2))
inspect(groceryrules[1:3])
```

Although running the `apriori()` function is straightforward, there can sometimes be a fair amount of trial and error needed to find the `support` and `confidence` parameters that produce a reasonable number of association rules. If you set these levels too high, you might find no rules or rules that are too generic to be very useful. On the other hand, a threshold too low might result in an unwieldy number of rules, or worse, the operation might take a very long time or run out of memory during the learning phase.

In this case, if we attempt to use the default settings of `support = 0.1` and `confidence = 0.8`, we will end up with a set of zero rules:

```
> apriori(groceries)
set of 0 rules
```

Obviously, we need to widen the search a bit.

If you think about it, this outcome should not have been terribly surprising. Because `support = 0.1` by default, in order to generate a rule, an item must have appeared in at least *0.1 * 9,385 = 938.5* transactions. Since only eight items appeared this frequently in our data, it's no wonder that we didn't find any rules.

One way to approach the problem of setting a minimum support threshold is to think about the smallest number of transactions you would need before you would consider a pattern interesting. For instance, you could argue that if an item is purchased twice a day (about 60 times in a month of data), it may be an interesting pattern. From there, it is possible to calculate the support level needed to find only the rules matching at least that many transactions. Since 60 out of 9,835 equals 0.006, we'll try setting the support there first.

Setting the minimum confidence involves a delicate balance. On one hand, if confidence is too low, we might be overwhelmed with a large number of unreliable rules—such as dozens of rules indicating the items commonly purchased with batteries. How would we know where to target our advertising budget then? On the other hand, if we set confidence too high, we will be limited to the rules that are obvious or inevitable—similar to the fact that smoke detectors are always purchased in combination with batteries. In this case, moving the smoke detectors closer to the batteries is unlikely to generate additional revenue, since the two items already were almost always purchased together.

The appropriate minimum confidence level depends a great deal on the goals of your analysis. If you start with a conservative value, you can always reduce it to broaden the search if you aren't finding actionable intelligence.

We'll start with a confidence threshold of 0.25, which means that in order to be included in the results, the rule has to be correct at least 25 percent of the time. This will eliminate the most unreliable rules, while allowing some room for us to modify behavior with targeted promotions.

We are now ready to generate some rules. In addition to the minimum `support` and `confidence` parameters, it is helpful to set `minlen = 2` to eliminate rules that contain fewer than two items. This prevents uninteresting rules from being created simply because the item is purchased frequently, for instance, *{} → whole milk*. This rule meets the minimum support and confidence because whole milk is purchased in over 25 percent of the transactions, but it isn't a very actionable insight.

The full command to find a set of association rules using the Apriori algorithm is as follows:

```
> groceryrules <- apriori(groceries, parameter = list(support =
                          0.006, confidence = 0.25, minlen = 2))
```

This saves our rules in a `rules` object, can take a peek into by typing its name:

```
> groceryrules
set of 463 rules
```

Our `groceryrules` object contains a set of 463 association rules. To determine whether any of them are useful, we'll have to dig deeper.

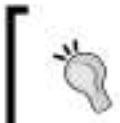
Step 4 – evaluating model performance

To obtain a high-level overview of the association rules, we can use `summary()` as follows. The rule length distribution tells us how many rules have each count of items. In our rule set, 150 rules have only two items, while 297 have three, and 16 have four. The `summary` statistics associated with this distribution are also given:

```
> summary(groceryrules)
set of 463 rules

rule length distribution (lhs + rhs):sizes
  2   3   4
150 297  16

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.000   2.000   3.000   2.711   3.000   4.000
```



As noted in the previous output, the size of the rule is calculated as the total of both the left-hand side (`lhs`) and right-hand side (`rhs`) of the rule. This means that a rule like *{bread} → {butter}* is two items and *{peanut butter, jelly} → {bread}* is three.

Next, we see the summary statistics of the rule quality measures: support, confidence, and lift. The support and confidence measures should not be very surprising, since we used these as selection criteria for the rules. We might be alarmed if most or all of the rules had support and confidence very near the minimum thresholds, as this would mean that we may have set the bar too high. This is not the case here, as there are many rules with much higher values of each.

```
summary of quality measures:
    support         confidence          lift
 Min.   :0.006101   Min.   :0.2500   Min.   :0.9932
 1st Qu.:0.007117   1st Qu.:0.2971   1st Qu.:1.6229
```

```
Median :0.008744   Median :0.3554   Median :1.9332
Mean   :0.011539   Mean   :0.3786   Mean   :2.0351
3rd Qu.:0.012303   3rd Qu.:0.4495   3rd Qu.:2.3565
Max.   :0.074835   Max.   :0.6600   Max.   :3.9565
```

The third column is a metric we have not considered yet. The *lift* of a rule measures how much more likely one item or itemset is purchased relative to its typical rate of purchase, given that you know another item or itemset has been purchased. This is defined by the following equation:

$$\text{lift}(X \rightarrow Y) = \frac{\text{confidence}(X \rightarrow Y)}{\text{support}(Y)}$$

Unlike confidence where the item order matters, *lift(X → Y)* is the same as *lift(Y → X)*.

For example, suppose at a grocery store, most people purchase milk and bread. By chance alone, we would expect to find many transactions with both milk and bread. However, if *lift(milk → bread)* is greater than one, it implies that the two items are found together more often than one would expect by chance. A large lift value is therefore a strong indicator that a rule is important, and reflects a true connection between the items.

In the final section of the `summary()` output, we receive mining information, telling us about how the rules were chosen. Here, we see that the `groceries` data, which contained 9,835 transactions, was used to construct rules with a minimum support of 0.0006 and minimum confidence of 0.25:

```
mining info:
      data ntransactions support confidence
 groceries          9835   0.006       0.25
```

We can take a look at specific rules using the `inspect()` function. For instance, the first three rules in the `groceryrules` object can be viewed as follows:

```
> inspect(groceryrules[1:3])

    lhs                rhs                    support     confidence     lift
  1 {potted plants} => {whole milk}       0.006914082  0.4000000 1.565460
  2 {pasta}         => {whole milk}       0.006100661  0.4054054 1.586614
  3 {herbs}         => {root vegetables} 0.007015760  0.4312500 3.956477
```

The first rule can be read in plain language as, "if a customer buys potted plants, they will also buy whole milk." With support of 0.007 and confidence of 0.400, we can determine that this rule covers 0.7 percent of the transactions and is correct in 40 percent of purchases involving potted plants. The lift value tells us how much more likely a customer is to buy whole milk relative to the average customer, given that he or she bought a potted plant. Since we know that about 25.6 percent of the customers bought whole milk (`support`), while 40 percent of the customers buying a potted plant bought whole milk (`confidence`), we can compute the lift value as *0.40 / 0.256 = 1.56*, which matches the value shown.

Note that the column labeled `support` indicates the support value for the rule, not the support value for the `lhs` or `rhs` alone).

In spite of the fact that the confidence and lift are high, does *{potted plants} → {whole milk}* seem like a very useful rule? Probably not, as there doesn't seem to be a logical reason why someone would be more likely to buy milk with a potted plant. Yet our data suggests otherwise. How can we make sense of this fact?

A common approach is to take the association rules and divide them into the following three categories:

- Actionable
- Trivial
- Inexplicable

Obviously, the goal of a market basket analysis is to find **actionable** rules that provide a clear and useful insight. Some rules are clear, others are useful; it is less common to find a combination of both of these factors.

So-called **trivial rules** include any rules that are so obvious that they are not worth mentioning—they are clear, but not useful. Suppose you were a marketing consultant being paid large sums of money to identify new opportunities for cross-promoting items. If you report the finding that *{diapers} → {formula}*, you probably won't be invited back for another consulting job.

Trivial rules can also sneak in disguised as more interesting results. For instance, say you found an association between a particular brand of children's cereal and a certain DVD movie. This finding is not very insightful if the movie's main character is on the front of the cereal box.

Rules are **inexplicable** if the connection between the items is so unclear that figuring out how to use the information is impossible or nearly impossible. The rule may simply be a random pattern in the data, for instance, a rule stating that *{pickles} → {chocolate ice cream}* may be due to a single customer, whose pregnant wife had regular cravings for strange combinations of foods.

The best rules are hidden gems—those undiscovered insights into patterns that seem obvious once discovered. Given enough time, one could evaluate each and every rule to find the gems. However, we (the one performing the market basket analysis) may not be the best judge of whether a rule is actionable, trivial, or inexplicable. In the next section, we'll improve the utility of our work by employing methods to sort and share the learned rules so that the most interesting results might float to the top.

Step 5 – improving model performance

Subject matter experts may be able to identify useful rules very quickly, but it would be a poor use of their time to ask them to evaluate hundreds or thousands of rules. Therefore, it's useful to be able to sort rules according to different criteria, and get them out of R into a form that can be shared with marketing teams and examined in more depth. In this way, we can improve the performance of our rules by making the results more actionable.

Sorting the set of association rules

Depending upon the objectives of the market basket analysis, the most useful rules might be the ones with the highest `support`, `confidence`, or `lift`. The `arules` package includes a `sort()` function that can be used to reorder the list of rules so that the ones with the highest or lowest values of the quality measure come first.

To reorder the `groceryrules` object, we can apply `sort()` while specifying a `"support"`, `"confidence"`, or `"lift"` value to the by parameter. By combining the `sort` function with vector operators, we can obtain a specific number of interesting rules. For instance, the best five rules according to the lift statistic can be examined using the following command:

```
> inspect(sort(groceryrules, by = "lift")[1:5])
```

These output is shown as follows:

```
  lhs                     rhs                     support confidence     lift
1 {herbs}              => {root vegetables}    0.007015760  0.4312500 3.956477
2 {berries}            => {whipped/sour cream} 0.009049314  0.2721713 3.796886
3 {other vegetables,
   tropical fruit,
   whole milk}         => {root vegetables}    0.007015760  0.4107143 3.768074
4 {beef,
   other vegetables}   => {root vegetables}    0.007930859  0.4020619 3.688692
5 {other vegetables,
   tropical fruit}     => {pip fruit}          0.009456024  0.2634561 3.482649
```

These rules appear to be more interesting than the ones we looked at previously. The first rule, with a `lift` of about 3.96, implies that people who buy herbs are nearly four times more likely to buy root vegetables than the typical customer—perhaps for a stew of some sort? Rule two is also interesting. Whipped cream is over three times more likely to be found in a shopping cart with berries versus other carts, suggesting perhaps a dessert pairing?

By default, the sort order is decreasing, meaning the largest values come first. To reverse this order, add an additional line, `parameterdecreasing = FALSE`.

Taking subsets of association rules

Suppose that given the preceding rule, the marketing team is excited about the possibilities of creating an advertisement to promote berries, which are now in season. Before finalizing the campaign, however, they ask you to investigate whether berries are often purchased with other items. To answer this question, we'll need to find all the rules that include berries in some form.

The `subset()` function provides a method to search for subsets of transactions, items, or rules. To use it to find any rules with `berries` appearing in the rule, use the following command. It will store the rules in a new object titled `berryrules`:

```
> berryrules <- subset(groceryrules, items %in% "berries")
```

We can then inspect the rules as we did with the larger set:

```
> inspect(berryrules)
```

The result is the following set of rules:

```
  lhs          rhs                      support confidence     lift
1 {berries} => {whipped/sour cream} 0.009049314  0.2721713 3.796886
2 {berries} => {yogurt}             0.010574479  0.3180428 2.279848
3 {berries} => {other vegetables}   0.010269446  0.3088685 1.596280
4 {berries} => {whole milk}         0.011794611  0.3547401 1.388328
```

There are four rules involving berries, two of which seem to be interesting enough to be called actionable. In addition to whipped cream, berries are also purchased frequently with yogurt—a pairing that could serve well for breakfast or lunch as well as dessert.

The `subset()` function is very powerful. The criteria for choosing the subset can be defined with several keywords and operators:

- The keyword `items` explained previously, matches an item appearing anywhere in the rule. To limit the subset to where the match occurs only on the left- or right-hand side, use `lhs` and `rhs` instead.
- The operator `%in%` means that at least one of the items must be found in the list you defined. If you want any rules matching either berries or yogurt, you could write `items %in%c("berries", "yogurt")`.
- Additional operators are available for partial matching (`%pin%`) and complete matching (`%ain%`). Partial matching allows you to find both citrus fruit and tropical fruit using one search: `items %pin% "fruit"`. Complete matching requires that all the listed items are present. For instance, `items %ain% c("berries", "yogurt")` finds only rules with both `berries` and `yogurt`.
- Subsets can also be limited by `support`, `confidence`, or `lift`. For instance, `confidence > 0.50` would limit you to the rules with confidence greater than 50 percent.
- Matching criteria can be combined with the standard R logical operators such as and (`&`), or (`|`), and not (`!`).

Using these options, you can limit the selection of rules to be as specific or general as you would like.

Saving association rules to a file or data frame

To share the results of your market basket analysis, you can save the rules to a CSV file with the `write()` function. This will produce a CSV file that can be used in most spreadsheet programs including Microsoft Excel:

```
> write(groceryrules, file = "groceryrules.csv",
        sep = ",", quote = TRUE, row.names = FALSE)
```

Sometimes it is also convenient to convert the rules into an R data frame. This can be accomplished easily using the `as()` function, as follows:

```
> groceryrules_df <- as(groceryrules, "data.frame")
```

This creates a data frame with the rules in the factor format, and numeric vectors for `support`, `confidence`, and `lift`:

```
> str(groceryrules_df)
'data.frame':463 obs. of 4 variables:
 $ rules      : Factor w/ 463 levels "{baking powder} => {other
vegetables}",..: 340 302 207 206 208 341 402 21 139 140 ...
 $ support    : num  0.00691 0.0061 0.00702 0.00773 0.00773 ...
 $ confidence: num  0.4 0.405 0.431 0.475 0.475 ...
 $ lift       : num  1.57 1.59 3.96 2.45 1.86 ...
```

You might choose to do this if you want to perform additional processing on the rules or need to export them to another database.

Summary

Association rules are frequently used to find provide useful insights in the massive transaction databases of large retailers. As an unsupervised learning process, association rule learners are capable of extracting knowledge from large databases without any prior knowledge of what patterns to seek. The catch is that it takes some effort to reduce the wealth of information into a smaller and more manageable set of results. The Apriori algorithm, which we studied in this chapter, does so by setting minimum thresholds of interestingness, and reporting only the associations meeting these criteria.

We put the Apriori algorithm to work while performing a market basket analysis for a month's worth of transactions at a moderately sized supermarket. Even in this small example, a wealth of associations was identified. Among these, we noted several patterns that may be useful for future marketing campaigns. The same methods we applied are used at much larger retailers on databases many times this size.

In the next chapter, we will examine another unsupervised learning algorithm. Much like association rules, it is intended to find patterns within data. But unlike association rules that seek patterns within the features, the methods in the next chapter are concerned with finding connections among the examples.

9

Finding Groups of Data – Clustering with k-means

Have you ever spent time watching a large crowd? If so, you are likely to have seen some recurring personalities. Perhaps a certain type of person, identified by a freshly pressed suit and a briefcase, comes to typify the "fat cat" business executive. A twenty-something wearing skinny jeans, a flannel shirt, and sunglasses might be dubbed a "hipster," while a woman unloading children from a minivan may be labeled a "soccer mom."

Of course, these types of stereotypes are dangerous to apply to individuals, as no two people are exactly alike. Yet understood as a way to describe a collective, the labels capture some underlying aspect of similarity among the individuals within the group.

As you will soon learn, the act of clustering, or spotting patterns in data, is not much different from spotting patterns in groups of people. In this chapter, you will learn:

- The ways clustering tasks differ from the classification tasks we examined previously
- How clustering defines a group, and how such groups are identified by k-means, a classic and easy-to-understand clustering algorithm
- The steps needed to apply clustering to a real-world task of identifying marketing segments among teenage social media users

Before jumping into action, we'll begin by taking an in-depth look at exactly what clustering entails.

Understanding clustering

Clustering is an unsupervised machine learning task that automatically divides the data into **clusters**, or groups of similar items. It does this without having been told how the groups should look ahead of time. As we may not even know what we're looking for, clustering is used for knowledge discovery rather than prediction. It provides an insight into the natural groupings found within data.

Without advance knowledge of what comprises a cluster, how can a computer possibly know where one group ends and another begins? The answer is simple. Clustering is guided by the principle that items inside a cluster should be very similar to each other, but very different from those outside. The definition of similarity might vary across applications, but the basic idea is always the same—group the data so that the related elements are placed together.

The resulting clusters can then be used for action. For instance, you might find clustering methods employed in the following applications:

- Segmenting customers into groups with similar demographics or buying patterns for targeted marketing campaigns
- Detecting anomalous behavior, such as unauthorized network intrusions, by identifying patterns of use falling outside the known clusters
- Simplifying extremely large datasets by grouping features with similar values into a smaller number of homogeneous categories

Overall, clustering is useful whenever diverse and varied data can be exemplified by a much smaller number of groups. It results in meaningful and actionable data structures that reduce complexity and provide insight into patterns of relationships.

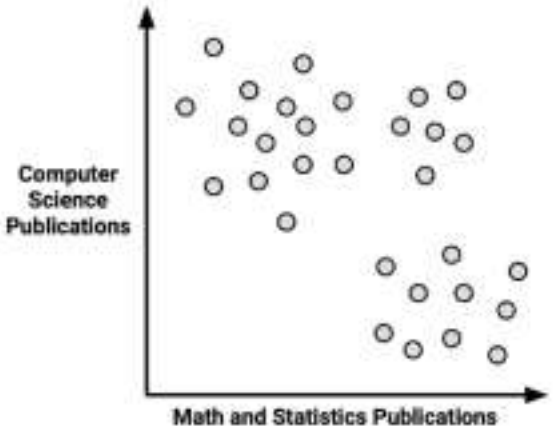
Clustering as a machine learning task

Clustering is somewhat different from the classification, numeric prediction, and pattern detection tasks we examined so far. In each of these cases, the result is a model that relates features to an outcome or features to other features; conceptually, the model describes the existing patterns within data. In contrast, clustering creates new data. Unlabeled examples are given a cluster label that has been inferred entirely from the relationships within the data. For this reason, you will, sometimes, see the clustering task referred to as **unsupervised classification** because, in a sense, it classifies unlabeled examples.

The catch is that the class labels obtained from an unsupervised classifier are without intrinsic meaning. Clustering will tell you which groups of examples are closely related—for instance, it might return the groups A, B, and C—but it's up to you to apply an actionable and meaningful label. To see how this impacts the clustering task, let's consider a hypothetical example.

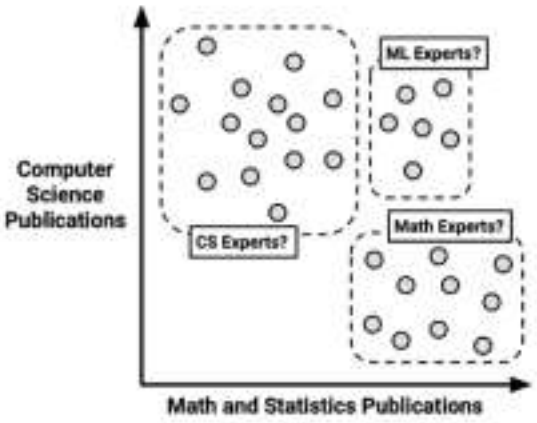
Suppose you were organizing a conference on the topic of data science. To facilitate professional networking and collaboration, you planned to seat people in groups according to one of three research specialties: computer and/or database science, math and statistics, and machine learning. Unfortunately, after sending out the conference invitations, you realize that you had forgotten to include a survey asking which discipline the attendee would prefer to be seated with.

In a stroke of brilliance, you realize that you might be able to infer each scholar's research specialty by examining his or her publication history. To this end, you begin collecting data on the number of articles each attendee published in computer science-related journals and the number of articles published in math or statistics-related journals. Using the data collected for several scholars, you create a scatterplot:



As expected, there seems to be a pattern. We might guess that the upper-left corner, which represents people with many computer science publications but few articles on math, could be a cluster of computer scientists. Following this logic, the lower-right corner might be a group of mathematicians. Similarly, the upper-right corner, those with both math and computer science experience, may be machine learning experts.

Our groupings were formed visually; we simply identified clusters as closely grouped data points. Yet in spite of the seemingly obvious groupings, we unfortunately have no way to know whether they are truly homogeneous without personally asking each scholar about his/her academic specialty. The labels we applied required us to make qualitative, presumptive judgments about the types of people that would fall into the group. For this reason, you might imagine the cluster labels in uncertain terms, as follows:



Rather than defining the group boundaries subjectively, it would be nice to use machine learning to define them objectively. Given the axis-parallel splits in the preceding diagram, our problem seems like an obvious application for the decision trees described in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*. This might provide us with a rule in the form "if a scholar has few math publications, then he/she is a computer science expert." Unfortunately, there's a problem with this plan. As we do not have data on the true class value for each point, a supervised learning algorithm would have no ability to learn such a pattern, as it would have no way of knowing what splits would result in homogenous groups.

On the other hand, clustering algorithms use a process very similar to what we did by visually inspecting the scatterplot. Using a measure of how closely the examples are related, homogeneous groups can be identified. In the next section, we'll start looking at how clustering algorithms are implemented.

This example highlights an interesting application of clustering. If you begin with unlabeled data, you can use clustering to create class labels. From there, you could apply a supervised learner such as decision trees to find the most important predictors of these classes. This is called **semi-supervised learning**.

The k-means clustering algorithm

The **k-means algorithm** is perhaps the most commonly used clustering method. Having been studied for several decades, it serves as the foundation for many more sophisticated clustering techniques. If you understand the simple principles it uses, you will have the knowledge needed to understand nearly any clustering algorithm in use today. Many such methods are listed on the following site, the **CRAN Task View** for clustering at `http://cran.r-project.org/web/views/Cluster.html`.

As k-means has evolved over time, there are many implementations of the algorithm. One popular approach is described in : Hartigan JA, Wong MA. A k-means clustering algorithm. Applied *Statistics*. 1979; 28:100-108.

Even though clustering methods have advanced since the inception of k-means, this is not to imply that k-means is obsolete. In fact, the method may be more popular now than ever. The following table lists some reasons why k-means is still used widely:

Strengths	Weaknesses
• Uses simple principles that can be explained in non-statistical terms • Highly flexible, and can be adapted with simple adjustments to address nearly all of its shortcomings • Performs well enough under many real-world use cases	• Not as sophisticated as more modern clustering algorithms • Because it uses an element of random chance, it is not guaranteed to find the optimal set of clusters • Requires a reasonable guess as to how many clusters naturally exist in the data • Not ideal for non-spherical clusters or clusters of widely varying density

If the name k-means sounds familiar to you, you may be recalling the k-NN algorithm discussed in *Chapter 3, Lazy Learning – Classification Using Nearest Neighbors*. As you will soon see, k-means shares more in common with the k-nearest neighbors than just the letter k.

The k-means algorithm assigns each of the *n* examples to one of the *k* clusters, where *k* is a number that has been determined ahead of time. The goal is to minimize the differences within each cluster and maximize the differences between the clusters.

Unless k and n are extremely small, it is not feasible to compute the optimal clusters across all the possible combinations of examples. Instead, the algorithm uses a heuristic process that finds **locally optimal** solutions. Put simply, this means that it starts with an initial guess for the cluster assignments, and then modifies the assignments slightly to see whether the changes improve the homogeneity within the clusters.

We will cover the process in depth shortly, but the algorithm essentially involves two phases. First, it assigns examples to an initial set of k clusters. Then, it updates the assignments by adjusting the cluster boundaries according to the examples that currently fall into the cluster. The process of updating and assigning occurs several times until changes no longer improve the cluster fit. At this point, the process stops and the clusters are finalized.

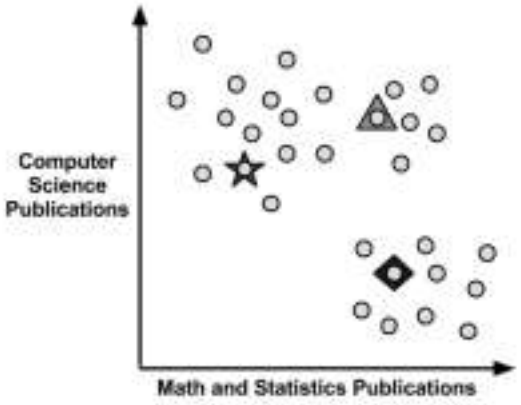
Due to the heuristic nature of k-means, you may end up with somewhat different final results by making only slight changes to the starting conditions. If the results vary dramatically, this could indicate a problem. For instance, the data may not have natural groupings or the value of k has been poorly chosen. With this in mind, it's a good idea to try a cluster analysis more than once to test the robustness of your findings.

To see how the process of assigning and updating works in practice, let's revisit the case of the hypothetical data science conference. Though this is a simple example, it will illustrate the basics of how k-means operates under the hood.

Using distance to assign and update clusters

As with k-NN, k-means treats feature values as coordinates in a multidimensional feature space. For the conference data, there are only two features, so we can represent the feature space as a two-dimensional scatterplot as depicted previously.

The k-means algorithm begins by choosing k points in the feature space to serve as the cluster centers. These centers are the catalyst that spurs the remaining examples to fall into place. Often, the points are chosen by selecting k random examples from the training dataset. As we hope to identify three clusters, according to this method, $k = 3$ points will be selected at random. These points are indicated by the star, triangle, and diamond in the following diagram:



It's worth noting that although the three cluster centers in the preceding diagram happen to be widely spaced apart, this is not always necessarily the case. Since they are selected at random, the three centers could have just as easily been three adjacent points. As the k-means algorithm is highly sensitive to the starting position of the cluster centers, this means that random chance may have a substantial impact on the final set of clusters.

To address this problem, k-means can be modified to use different methods for choosing the initial centers. For example, one variant chooses random values occurring anywhere in the feature space (rather than only selecting among the values observed in the data). Another option is to skip this step altogether; by randomly assigning each example to a cluster, the algorithm can jump ahead immediately to the update phase. Each of these approaches adds a particular bias to the final set of clusters, which you may be able to use to improve your results.

In 2007, an algorithm called **k-means++** was introduced, which proposes an alternative method for selecting the initial cluster centers. It purports to be an efficient way to get much closer to the optimal clustering solution while reducing the impact of random chance. For more information, refer to *Arthur D, Vassilvitskii S.* k-means++: The advantages of careful seeding. *Proceedings of the eighteenth annual ACM-SIAM symposium on discrete algorithms.* 2007:1027–1035.

After choosing the initial cluster centers, the other examples are assigned to the cluster center that is nearest according to the distance function. You will remember that we studied distance functions while learning about k-Nearest Neighbors. Traditionally, k-means uses Euclidean distance, but Manhattan distance or Minkowski distance are also sometimes used.

Recall that if n indicates the number of features, the formula for Euclidean distance between example x and example y is:

$$\text{dist}(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

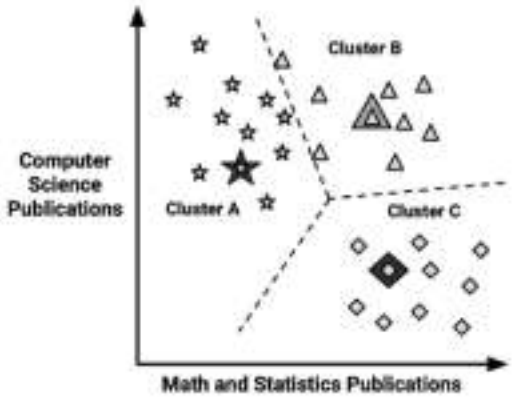
For instance, if we are comparing a guest with five computer science publications and one math publication to a guest with zero computer science papers and two math papers, we could compute this in R as follows:

```
> sqrt((5 - 0)^2 + (1 - 2)^2)
[1] 5.09902
```

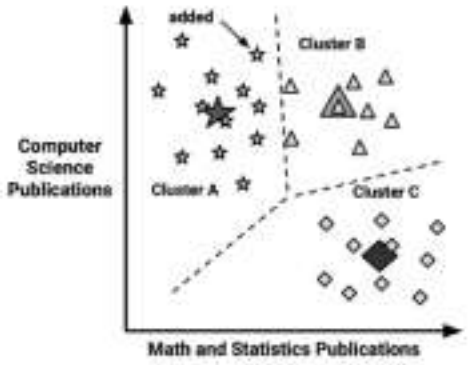
Using this distance function, we find the distance between each example and each cluster center. The example is then assigned to the nearest cluster center.

Keep in mind that as we are using distance calculations, all the features need to be numeric, and the values should be normalized to a standard range ahead of time. The methods discussed in *Chapter 3, Lazy Learning – Classification Using Nearest Neighbors*, will prove helpful for this task.

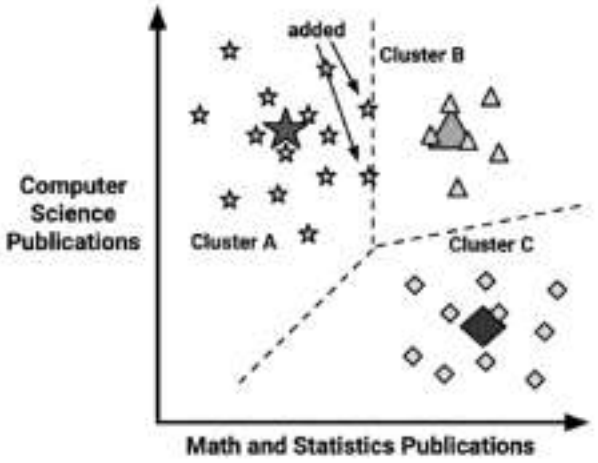
As shown in the following diagram, the three cluster centers partition the examples into three segments labeled **Cluster A**, **Cluster B**, and **Cluster C**. The dashed lines indicate the boundaries for the **Voronoi diagram** created by the cluster centers. The Voronoi diagram indicates the areas that are closer to one cluster center than any other; the vertex where all the three boundaries meet is the maximal distance from all three cluster centers. Using these boundaries, we can easily see the regions claimed by each of the initial k-means seeds:



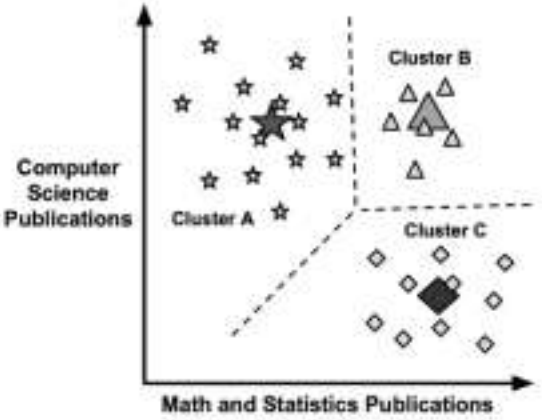
Now that the initial assignment phase has been completed, the k-means algorithm proceeds to the update phase. The first step of updating the clusters involves shifting the initial centers to a new location, known as the **centroid**, which is calculated as the average position of the points currently assigned to that cluster. The following diagram illustrates how as the cluster centers shift to the new centroids, the boundaries in the Voronoi diagram also shift and a point that was once in **Cluster B** (indicated by an arrow) is added to **Cluster A**:



As a result of this reassignment, the k-means algorithm will continue through another update phase. After shifting the cluster centroids, updating the cluster boundaries, and reassigning points into new clusters (as indicated by arrows), the figure looks like this:



Because two more points were reassigned, another update must occur, which moves the centroids and updates the cluster boundaries. However, because these changes result in no reassignments, the k-means algorithm stops. The cluster assignments are now final:



The final clusters can be reported in one of the two ways. First, you might simply report the cluster assignments such as A, B, or C for each example. Alternatively, you could report the coordinates of the cluster centroids after the final update. Given either reporting method, you are able to define the cluster boundaries by calculating the centroids or assigning each example to its nearest cluster.

Choosing the appropriate number of clusters

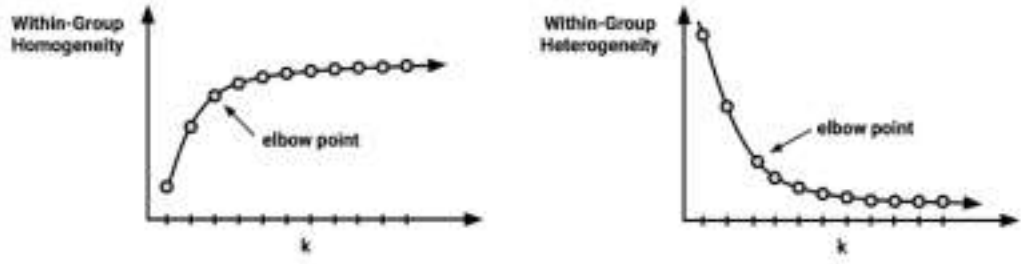
In the introduction to k-means, we learned that the algorithm is sensitive to the randomly-chosen cluster centers. Indeed, if we had selected a different combination of three starting points in the previous example, we may have found clusters that split the data differently from what we had expected. Similarly, k-means is sensitive to the number of clusters; the choice requires a delicate balance. Setting *k* to be very large will improve the homogeneity of the clusters, and at the same time, it risks overfitting the data.

Ideally, you will have *a priori* knowledge (a prior belief) about the true groupings and you can apply this information to choosing the number of clusters. For instance, if you were clustering movies, you might begin by setting *k* equal to the number of genres considered for the Academy Awards. In the data science conference seating problem that we worked through previously, *k* might reflect the number of academic fields of study that were invited.

Sometimes the number of clusters is dictated by business requirements or the motivation for the analysis. For example, the number of tables in the meeting hall could dictate how many groups of people should be created from the data science attendee list. Extending this idea to another business case, if the marketing department only has resources to create three distinct advertising campaigns, it might make sense to set $k = 3$ to assign all the potential customers to one of the three appeals.

Without any prior knowledge, one rule of thumb suggests setting k equal to the square root of $(n / 2)$, where n is the number of examples in the dataset. However, this rule of thumb is likely to result in an unwieldy number of clusters for large datasets. Luckily, there are other statistical methods that can assist in finding a suitable k-means cluster set.

A technique known as the **elbow method** attempts to gauge how the homogeneity or heterogeneity within the clusters changes for various values of k. As illustrated in the following diagrams, the homogeneity within clusters is expected to increase as additional clusters are added; similarly, heterogeneity will also continue to decrease with more clusters. As you could continue to see improvements until each example is in its own cluster, the goal is not to maximize homogeneity or minimize heterogeneity, but rather to find k so that there are diminishing returns beyond that point. This value of k is known as the **elbow point** because it looks like an elbow.



There are numerous statistics to measure homogeneity and heterogeneity within the clusters that can be used with the elbow method (the following information box provides a citation for more detail). Still, in practice, it is not always feasible to iteratively test a large number of k values. This is in part because clustering large datasets can be fairly time consuming; clustering the data repeatedly is even worse. Regardless, applications requiring the exact optimal set of clusters are fairly rare. In most clustering applications, it suffices to choose a k value based on convenience rather than strict performance requirements.

For a very thorough review of the vast assortment of cluster performance measures, refer to: *Halkidi M, Batistakis Y, Vazirgiannis M.* On clustering validation techniques. *Journal of Intelligent Information Systems*. 2001; 17:107-145.

The process of setting *k* itself can sometimes lead to interesting insights. By observing how the characteristics of the clusters change as *k* is varied, one might infer where the data have naturally defined boundaries. Groups that are more tightly clustered will change a little, while less homogeneous groups will form and disband over time.

In general, it may be wise to spend little time worrying about getting *k* exactly right. The next example will demonstrate how even a tiny bit of subject-matter knowledge borrowed from a Hollywood film can be used to set *k* such that actionable and interesting clusters are found. As clustering is unsupervised, the task is really about what you make of it; the value is in the insights you take away from the algorithm's findings.

Example – finding teen market segments using k-means clustering

Interacting with friends on a **social networking service** (SNS), such as Facebook, Tumblr, and Instagram has become a rite of passage for teenagers around the world. Having a relatively large amount of disposable income, these adolescents are a coveted demographic for businesses hoping to sell snacks, beverages, electronics, and hygiene products.

The many millions of teenage consumers using such sites have attracted the attention of marketers struggling to find an edge in an increasingly competitive market. One way to gain this edge is to identify segments of teenagers who share similar tastes, so that clients can avoid targeting advertisements to teens with no interest in the product being sold. For instance, sporting apparel is likely to be a difficult sell to teens with no interest in sports.

Given the text of teenagers' SNS pages, we can identify groups that share common interests such as sports, religion, or music. Clustering can automate the process of discovering the natural segments in this population. However, it will be up to us to decide whether or not the clusters are interesting and how we can use them for advertising. Let's try this process from start to finish.

Step 1 – collecting data

For this analysis, we will use a dataset representing a random sample of 30,000 U.S. high school students who had profiles on a well-known SNS in 2006. To protect the users' anonymity, the SNS will remain unnamed. However, at the time the data was collected, the SNS was a popular web destination for US teenagers. Therefore, it is reasonable to assume that the profiles represent a fairly wide cross section of American adolescents in 2006.

This dataset was compiled by Brett Lantz while conducting sociological research on the teenage identities at the University of Notre Dame. If you use the data for research purposes, please cite this book chapter. The full dataset is available at the Packt Publishing website with the filename `snsdata.csv`. To follow along interactively, this chapter assumes that you have saved this file to your R working directory.

The data was sampled evenly across four high school graduation years (2006 through 2009) representing the senior, junior, sophomore, and freshman classes at the time of data collection. Using an automated web crawler, the full text of the SNS profiles were downloaded, and each teen's gender, age, and number of SNS friends was recorded.

A text mining tool was used to divide the remaining SNS page content into words. From the top 500 words appearing across all the pages, 36 words were chosen to represent five categories of interests: namely extracurricular activities, fashion, religion, romance, and antisocial behavior. The 36 words include terms such as *football*, *sexy*, *kissed*, *bible*, *shopping*, *death*, and *drugs*. The final dataset indicates, for each person, how many times each word appeared in the person's SNS profile.

Step 2 – exploring and preparing the data

We can use the default settings of `read.csv()` to load the data into a data frame:

```
> teens <- read.csv("snsdata.csv")
```

Let's also take a quick look at the specifics of the data. The first several lines of the `str()` output are as follows:

```
> str(teens)
'data.frame':  30000 obs. of  40 variables:
 $ gradyear    : int  2006 2006 2006 2006 2006 2006 2006 2006 ...
 $ gender      : Factor w/ 2 levels "F","M": 2 1 2 1 NA 1 1 2 ...
 $ age         : num  19 18.8 18.3 18.9 19 ...
```

```
 $ friends     : int  7 0 69 0 10 142 72 17 52 39 ...
 $ basketball  : int  0 0 0 0 0 0 0 0 0 0 ...
```

As we had expected, the data include 30,000 teenagers with four variables indicating personal characteristics and 36 words indicating interests.

Do you notice anything strange around the `gender` row? If you were looking carefully, you may have noticed the `NA` value, which is out of place compared to the `1` and `2` values. The `NA` is R's way of telling us that the record has a missing value—we do not know the person's gender. Until now, we haven't dealt with missing data, but it can be a significant problem for many types of analyses.

Let's see how substantial this problem is. One option is to use the `table()` command, as follows:

```
> table(teens$gender)
    F     M
22054  5222
```

Although this command tells us how many `F` and `M` values are present, the `table()` function excluded the `NA` values rather than treating it as a separate category. To include the `NA` values (if there are any), we simply need to add an additional parameter:

```
> table(teens$gender, useNA = "ifany")
    F     M  <NA>
22054  5222  2724
```

Here, we see that 2,724 records (9 percent) have missing gender data. Interestingly, there are over four times as many females as males in the SNS data, suggesting that males are not as inclined to use SNS websites as females.

If you examine the other variables in the data frame, you will find that besides `gender`, only `age` has missing values. For numeric data, the `summary()` command tells us the number of missing `NA` values:

```
> summary(teens$age)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  3.086  16.310  17.290  17.990  18.260 106.900    5086
```

A total of 5,086 records (17 percent) have missing ages. Also concerning is the fact that the minimum and maximum values seem to be unreasonable; it is unlikely that a 3 year old or a 106 year old is attending high school. To ensure that these extreme values don't cause problems for the analysis, we'll need to clean them up before moving on.

A more reasonable range of ages for the high school students includes those who are at least 13 years old and not yet 20 years old. Any age value falling outside this range should be treated the same as missing data—we cannot trust the age provided. To recode the age variable, we can use the `ifelse()` function, assigning `teen$age` the value of `teen$age` if the age is at least 13 and less than 20 years; otherwise, it will receive the value `NA`:

```
> teens$age <- ifelse(teens$age >= 13 & teens$age < 20,
                      teens$age, NA)
```

By rechecking the `summary()` output, we see that the age range now follows a distribution that looks much more like an actual high school:

```
> summary(teens$age)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  13.03   16.30   17.26   17.25   18.22   20.00    5523
```

Unfortunately, now we've created an even larger missing data problem. We'll need to find a way to deal with these values before continuing with our analysis.

Data preparation – dummy coding missing values

An easy solution for handling the missing values is to exclude any record with a missing value. However, if you think through the implications of this practice, you might think twice before doing so—just because it is easy does not mean it is a good idea! The problem with this approach is that even if the missingness is not extensive, you can easily exclude large portions of the data.

For example, suppose that in our data, the people with the `NA` values for gender are completely different from those with missing age data. This would imply that by excluding those missing either gender or age, you would exclude 9% + 17% = 26% of the data, or over 7,500 records. And this is for missing data on only two variables! The larger the number of missing values present in a dataset, the more likely it is that any given record will be excluded. Fairly soon, you will be left with a tiny subset of data, or worse, the remaining records will be systematically different or non-representative of the full population.

An alternative solution for categorical variables like gender is to treat a missing value as a separate category. For instance, rather than limiting to female and male, we can add an additional category for the unknown gender. This allows us to utilize dummy coding, which was covered in *Chapter 3, Lazy Learning – Classification Using Nearest Neighbors*.

If you recall, dummy coding involves creating a separate binary (1 or 0) valued dummy variable for each level of a nominal feature except one, which is held out to serve as the reference group. The reason one category can be excluded is because its status can be inferred from the other categories. For instance, if someone is not female and not unknown gender, they must be male. Therefore, in this case, we need to only create dummy variables for female and unknown gender:

```
> teens$female <- ifelse(teens$gender == "F" &
                           !is.na(teens$gender), 1, 0)
> teens$no_gender <- ifelse(is.na(teens$gender), 1, 0)
```

As you might expect, the `is.na()` function tests whether gender is equal to `NA`. Therefore, the first statement assigns `teens$female` the value `1` if gender is equal to `F` and the gender is not equal to `NA`; otherwise, it assigns the value `0`. In the second statement, if `is.na()` returns `TRUE`, meaning the gender is missing, the `teens$no_gender` variable is assigned `1`; otherwise, it is assigned the value `0`. To confirm that we did the work correctly, let's compare our constructed dummy variables to the original `gender` variable:

```
> table(teens$gender, useNA = "ifany")
    F     M  <NA>
22054  5222  2724
> table(teens$female, useNA = "ifany")
    0     1
 7946 22054
> table(teens$no_gender, useNA = "ifany")
    0     1
27276  2724
```

The number of `1` values for `teens$female` and `teens$no_gender` matches the number of `F` and `NA` values, respectively, so we should be able to trust our work.

Data preparation – imputing the missing values

Next, let's eliminate the 5,523 missing age values. As age is numeric, it doesn't make sense to create an additional category for the unknown values—where would you rank "unknown" relative to the other ages? Instead, we'll use a different strategy known as **imputation**, which involves filling in the missing data with a guess as to the true value.

Can you think of a way we might be able to use the SNS data to make an informed guess about a teenager's age? If you are thinking of using the graduation year, you've got the right idea. Most people in a graduation cohort were born within a single calendar year. If we can identify the typical age for each cohort, we would have a fairly reasonable estimate of the age of a student in that graduation year.

One way to find a typical value is by calculating the average or mean value. If we try to apply the `mean()` function, as we did for previous analyses, there's a problem:

```
> mean(teens$age)
[1] NA
```

The issue is that the mean is undefined for a vector containing missing data. As our age data contains missing values, `mean(teens$age)` returns a missing value. We can correct this by adding an additional parameter to remove the missing values before calculating the mean:

```
> mean(teens$age, na.rm = TRUE)
[1] 17.25243
```

This reveals that the average student in our data is about 17 years old. This only gets us part of the way there; we actually need the average age for each graduation year. You might be tempted to calculate the mean four times, but one of the benefits of R is that there's usually a way to avoid repeating oneself. In this case, the `aggregate()` function is the tool for the job. It computes statistics for subgroups of data. Here, it calculates the mean age by graduation year after removing the `NA` values:

```
> aggregate(data = teens, age ~ gradyear, mean, na.rm = TRUE)
  gradyear      age
1     2006 18.65586
2     2007 17.70617
3     2008 16.76770
4     2009 15.81957
```

The mean age differs by roughly one year per change in graduation year. This is not at all surprising, but a helpful finding for confirming our data is reasonable.

The `aggregate()` output is a data frame. This is helpful for some purposes, but would require extra work to merge back onto our original data. As an alternative, we can use the `ave()` function, which returns a vector with the group means repeated so that the result is equal in length to the original vector:

```
> ave_age <- ave(teens$age, teens$gradyear, FUN =
                 function(x) mean(x, na.rm = TRUE))
```

To impute these means onto the missing values, we need one more `ifelse()` call to use the `ave_age` value only if the original age value was `NA`:

```
> teens$age <- ifelse(is.na(teens$age), ave_age, teens$age)
```

The `summary()` results show that the missing values have now been eliminated:

```
> summary(teens$age)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  13.03   16.28   17.24   17.24   18.21   20.00
```

With the data ready for analysis, we are ready to dive into the interesting part of this project. Let's see whether our efforts have paid off.

Step 3 – training a model on the data

To cluster the teenagers into marketing segments, we will use an implementation of k-means in the `stats` package, which should be included in your R installation by default. If by chance you do not have this package, you can install it as you would any other package and load it using the `library(stats)` command. Although there is no shortage of k-means functions available in various R packages, the `kmeans()` function in the `stats` package is widely used and provides a vanilla implementation of the algorithm.

Clustering syntax

using the `kmeans()` function in the `stats` package

Finding clusters:

```
myclusters <- kmeans(mydata, k)
```

- `mydata` is a matrix or data frame with the examples to be clustered
- `k` specifies the desired number of clusters

The function will return a cluster object that stores information about the clusters.

Examining clusters:

- `myclusters$cluster` is a vector of cluster assignments from the `kmeans()` function
- `myclusters$centers` is a matrix indicating the mean values for each feature and cluster combination
- `myclusters$size` lists the number of examples assigned to each cluster

Example:

```
teen_clusters <- kmeans(teens, 5)
teens$cluster_id <- teen_clusters$cluster
```

The `kmeans()` function requires a data frame containing only numeric data and a parameter specifying the desired number of clusters. If you have these two things ready, the actual process of building the model is simple. The trouble is that choosing the right combination of data and clusters can be a bit of an art; sometimes a great deal of trial and error is involved.

We'll start our cluster analysis by considering only the 36 features that represent the number of times various interests appeared on the teen SNS profiles. For convenience, let's make a data frame containing only these features:

```
> interests <- teens[5:40]
```

If you recall from *Chapter 3, Lazy Learning – Classification Using Nearest Neighbors*, a common practice employed prior to any analysis using distance calculations is to normalize or z-score standardize the features so that each utilizes the same range. By doing so, you can avoid a problem in which some features come to dominate solely because they have a larger range of values than the others.

The process of z-score standardization rescales features so that they have a mean of zero and a standard deviation of one. This transformation changes the interpretation of the data in a way that may be useful here. Specifically, if someone mentions football three times on their profile, without additional information, we have no idea whether this implies they like football more or less than their peers. On the other hand, if the z-score is three, we know that that they mentioned football many more times than the average teenager.

To apply the z-score standardization to the `interests` data frame, we can use the `scale()` function with `lapply()` as follows:

```
> interests_z <- as.data.frame(lapply(interests, scale))
```

Since `lapply()` returns a matrix, it must be coerced back to data frame form using the `as.data.frame()` function.

Our last decision involves deciding how many clusters to use for segmenting the data. If we use too many clusters, we may find them too specific to be useful; conversely, choosing too few may result in heterogeneous groupings. You should feel comfortable experimenting with the values of *k*. If you don't like the result, you can easily try another value and start over.

Choosing the number of clusters is easier if you are familiar with the analysis population. Having a hunch about the true number of natural groupings can save you some trial and error.

To help us predict the number of clusters in the data, I'll defer to one of my favorite films, The *Breakfast Club*, a coming-of-age comedy released in 1985 and directed by John Hughes. The teenage characters in this movie are identified in terms of five stereotypes: a brain, an athlete, a basket case, a princess, and a criminal. Given that these identities prevail throughout popular teen fiction, five seems like a reasonable starting point for *k*.

To use the k-means algorithm to divide the teenagers' interest data into five clusters, we use the `kmeans()` function on the `interests` data frame. Because the k-means algorithm utilizes random starting points, the `set.seed()` function is used to ensure that the results match the output in the examples that follow. If you recall from the previous chapters, this command initializes R's random number generator to a specific sequence. In the absence of this statement, the results will vary each time the k-means algorithm is run:

```
> set.seed(2345)
> teen_clusters <- kmeans(interests_z, 5)
```

The result of the k-means clustering process is a list named `teen_clusters` that stores the properties of each of the five clusters. Let's dig in and see how well the algorithm has divided the teens' interest data.

If you find that your results differ from those shown here, ensure that the `set.seed(2345)` command is run immediately prior to the `kmeans()` function.

Step 4 – evaluating model performance

Evaluating clustering results can be somewhat subjective. Ultimately, the success or failure of the model hinges on whether the clusters are useful for their intended purpose. As the goal of this analysis was to identify clusters of teenagers with similar interests for marketing purposes, we will largely measure our success in qualitative terms. For other clustering applications, more quantitative measures of success may be needed.

One of the most basic ways to evaluate the utility of a set of clusters is to examine the number of examples falling in each of the groups. If the groups are too large or too small, they are not likely to be very useful. To obtain the size of the `kmeans()` clusters, use the `teen_clusters$size` component as follows:

```
> teen_clusters$size
[1]   871   600  5981  1034 21514
```

Here, we see the five clusters we requested. The smallest cluster has 600 teenagers (2 percent) while the largest cluster has 21,514 (72 percent). Although the large gap between the number of people in the largest and smallest clusters is slightly concerning, without examining these groups more carefully, we will not know whether or not this indicates a problem. It may be the case that the clusters' size disparity indicates something real, such as a big group of teens that share similar interests, or it may be a random fluke caused by the initial k-means cluster centers. We'll know more as we start to look at each cluster's homogeneity.

Sometimes, k-means may find extremely small clusters—occasionally, as small as a single point. This can happen if one of the initial cluster centers happens to fall on an outlier far from the rest of the data. It is not always clear whether to treat such small clusters as a true finding that represents a cluster of extreme cases, or a problem caused by random chance. If you encounter this issue, it may be worth re-running the k-means algorithm with a different random seed to see whether the small cluster is robust to different starting points.

For a more in-depth look at the clusters, we can examine the coordinates of the cluster centroids using the `teen_clusters$centers` component, which is as follows for the first four interests:

```
> teen_clusters$centers
   basketball   football      soccer    softball
1  0.16001227  0.2364174  0.10385512  0.07232021
2 -0.09195886  0.0652625 -0.09932124 -0.01739428
3  0.52755083  0.4873480  0.29778605  0.37178877
4  0.34081039  0.3593965  0.12722250  0.16384661
5 -0.16695523 -0.1641499 -0.09033520 -0.11367669
```

The rows of the output (labeled `1` to `5`) refer to the five clusters, while the numbers across each row indicate the cluster's average value for the interest listed at the top of the column. As the values are z-score standardized, positive values are above the overall mean level for all the teens and negative values are below the overall mean. For example, the third row has the highest value in the basketball column, which means that cluster `3` has the highest average interest in basketball among all the clusters.

By examining whether the clusters fall above or below the mean level for each interest category, we can begin to notice patterns that distinguish the clusters from each other. In practice, this involves printing the cluster centers and searching through them for any patterns or extreme values, much like a word search puzzle but with numbers. The following screenshot shows a highlighted pattern for each of the five clusters, for 19 of the 36 teen interests:

```
> teen_clusters$centers
   basketball   football      soccer    softball  volleyball    swimming
1  0.16001227  0.2364174  0.10385512  0.07232021  0.18897158  0.23970234
2 -0.09195886  0.0652625 -0.09932124 -0.01739428 -0.06219308  0.03339844
3  0.52755083  0.4873480  0.29778605  0.37178877  0.37986175  0.29628671
4  0.34081039  0.3593965  0.12722250  0.16384661  0.11032200  0.26943332
5 -0.16695523 -0.1641499 -0.09033520 -0.11367669 -0.11682181 -0.10595448
  cheerleading    baseball      tennis      sports        cute          sex
1    0.3931445  0.02993479  0.13532387  0.10257837  0.37884271  0.020042068
2   -0.1101103 -0.11487510  0.04062204 -0.09899231 -0.03265037 -0.042486141
3    0.3303485  0.35231971  0.14057808  0.32967130  0.54442929  0.002913623
4    0.1856664  0.27527088  0.10980958  0.79711920  0.47866008  2.028471066
5   -0.1136077 -0.10918483 -0.05097057 -0.13135334 -0.18878627 -0.097928345
         sexy         hot      kissed       dance        band    marching       music
1  0.11740551  0.41389104  0.06787768  0.22780899 -0.10257102 -0.10942590  0.1378306
2 -0.04329091 -0.03812345 -0.04554933  0.04573186  4.06726666  5.25757242  0.4981238
3  0.24040196  0.38551819 -0.03356121  0.45662534 -0.02120728 -0.10888541  0.2844999
4  0.51266080  0.31708549  2.97973077  0.45535061  0.38053621 -0.02014608  1.1367885
5 -0.09501817 -0.13810894 -0.13535855 -0.15932739 -0.12167214 -0.11098063 -0.1532006
```

Given this subset of the interest data, we can already infer some characteristics of the clusters. **Cluster 3** is substantially above the mean interest level on all the sports. This suggests that this may be a group of **Athletes** per *The Breakfast Club* stereotype. **Cluster 1** includes the most mentions of "cheerleading," the word "hot," and is above the average level of football interest. Are these the so-called **Princesses**?

By continuing to examine the clusters in this way, it is possible to construct a table listing the dominant interests of each of the groups. In the following table, each cluster is shown with the features that most distinguish it from the other clusters, and The *Breakfast Club* identity that most accurately captures the group's characteristics.

Interestingly, **Cluster 5** is distinguished by the fact that it is unexceptional; its members had lower-than-average levels of interest in every measured activity. It is also the single largest group in terms of the number of members. One potential explanation is that these users created a profile on the website but never posted any interests.

Cluster 1 (N = 3,376)	Cluster 2 (N = 601)	Cluster 3 (N = 1,036)	Cluster 4 (N = 3,279)	Cluster 5 (N = 21,708)
swimming cheerleading cute sexy hot dance dress hair mall hollister abercrombie shopping clothes	band marching music rock	sports sex sexy hot kissed dance music band die death drunk drugs	basketball football soccer softball volleyball baseball sports god church Jesus bible	???
Princesses	Brains	Criminals	Athletes	Basket Cases

When sharing the results of a segmentation analysis, it is often helpful to apply informative labels that simplify and capture the essence of the groups such as *The Breakfast Club* typology applied here. The risk in adding such labels is that they can obscure the groups' nuances by stereotyping the group members. As such labels can bias our thinking, important patterns can be missed if labels are taken as the whole truth.

Given the table, a marketing executive would have a clear depiction of five types of teenage visitors to the social networking website. Based on these profiles, the executive could sell targeted advertising impressions to businesses with products relevant to one or more of the clusters. In the next section, we will see how the cluster labels can be applied back to the original population for such uses.

Step 5 – improving model performance

Because clustering creates new information, the performance of a clustering algorithm depends at least somewhat on both the quality of the clusters themselves as well as what is done with that information. In the preceding section, we already demonstrated that the five clusters provided useful and novel insights into the interests of teenagers. By that measure, the algorithm appears to be performing quite well. Therefore, we can now focus our effort on turning these insights into action.

We'll begin by applying the clusters back onto the full dataset. The `teen_clusters` object created by the `kmeans()` function includes a component named `cluster` that contains the cluster assignments for all 30,000 individuals in the sample. We can add this as a column on the `teens` data frame with the following command:

```
> teens$cluster <- teen_clusters$cluster
```

Given this new data, we can start to examine how the cluster assignment relates to individual characteristics. For example, here's the personal information for the first five teens in the SNS data:

```
> teens[1:5, c("cluster", "gender", "age", "friends")]
  cluster gender    age friends
1       5      M 18.982       7
2       3      F 18.801       0
3       5      M 18.335      69
4       5      F 18.875       0
5       4   <NA> 18.995      10
```

Using the `aggregate()` function, we can also look at the demographic characteristics of the clusters. The mean age does not vary much by cluster, which is not too surprising as these teen identities are often determined before high school. This is depicted as follows:

```
> aggregate(data = teens, age ~ cluster, mean)
  cluster      age
1       1 16.86497
2       2 17.39037
3       3 17.07656
4       4 17.11957
5       5 17.29849
```

On the other hand, there are some substantial differences in the proportion of females by cluster. This is a very interesting finding as we didn't use gender data to create the clusters, yet the clusters are still predictive of gender:

```
> aggregate(data = teens, female ~ cluster, mean)
  cluster    female
1       1 0.8381171
2       2 0.7250000
3       3 0.8378198
4       4 0.8027079
5       5 0.6994515
```

Recall that overall about 74 percent of the SNS users are female. **Cluster 1**, the so-called **Princesses**, is nearly 84 percent female, while **Cluster 2** and **Cluster 5** are only about 70 percent female. These disparities imply that there are differences in the interests that teen boys and girls discuss on their social networking pages.

Given our success in predicting gender, you might also suspect that the clusters are predictive of the number of friends the users have. This hypothesis seems to be supported by the data, which is as follows:

```
> aggregate(data = teens, friends ~ cluster, mean)
  cluster  friends
1       1 41.43054
2       2 32.57333
3       3 37.16185
4       4 30.50290
5       5 27.70052
```

On an average, **Princesses** have the most friends (41.4), followed by **Athletes** (37.2) and **Brains** (32.6). On the low end are **Criminals** (30.5) and **Basket Cases** (27.7). As with gender, the connection between a teen's number of friends and their predicted cluster is remarkable, given that we did not use the friendship data as an input to the clustering algorithm. Also interesting is the fact that the number of friends seems to be related to the stereotype of each clusters' high school popularity; the stereotypically popular groups tend to have more friends.

The association among group membership, gender, and number of friends suggests that the clusters can be useful predictors of behavior. Validating their predictive ability in this way may make the clusters an easier sell when they are pitched to the marketing team, ultimately improving the performance of the algorithm.

Summary

Our findings support the popular adage that "birds of a feather flock together." By using machine learning methods to cluster teenagers with others who have similar interests, we were able to develop a typology of teen identities that was predictive of personal characteristics, such as gender and the number of friends. These same methods can be applied to other contexts with similar results.

This chapter covered only the fundamentals of clustering. As a very mature machine learning method, there are many variants of the k-means algorithm as well as many other clustering algorithms that bring unique biases and heuristics to the task. Based on the foundation in this chapter, you will be able to understand and apply other clustering methods to new problems.

In the next chapter, we will begin to look at methods for measuring the success of a learning algorithm, which are applicable across many machine learning tasks. While our process has always devoted some effort to evaluating the success of learning, in order to obtain the highest degree of performance, it is crucial to be able to define and measure it in the strictest terms.