

7

Black Box Methods – Neural Networks and Support Vector Machines

The late science fiction author *Arthur C. Clarke* once wrote that "any sufficiently advanced technology is indistinguishable from magic." This chapter covers a pair of machine learning methods that may, likewise, appear at first glance to be magic. As two of the most powerful machine learning algorithms, they are applied to tasks across many domains. However, 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 a figurative box. The reasons for the opacity can vary; for instance, black box closed source software intentionally conceals proprietary algorithms, the black box of sausage-making involves a bit of purposeful (but tasty) ignorance, and the black box of political lawmaking is rooted in bureaucratic processes. In the case of machine learning, the black box is because the underlying models are based on complex mathematical systems and the results are difficult to interpret.

Although it may not be feasible to interpret black box models, it is dangerous to apply the methods blindly. Therefore, in this chapter, we'll peek behind the curtain and investigate the statistical sausage-making involved in fitting such models. You'll discover that:

- Neural networks use concepts borrowed from an understanding of human brains in order to model arbitrary functions
- Support Vector Machines use multidimensional surfaces to define the relationship between features and outcomes
- In spite of their complexity, these models can be easily applied to real-world problems such as modeling the strength of concrete or reading printed text

With any luck, you'll realize that you don't need a black belt in statistics to tackle black box machine 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, the 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 storing 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 ANN state-of-the-art.



Though it may be infeasible to completely model a cockroach's brain, a neural network might provide an adequate heuristic model of its behavior, such as in an algorithm that can mimic how a roach flees when discovered. If the behavior of a roboroach is convincing, does it matter how its brain works? This question is the basis of the controversial **Turing test**, 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. These early exercises were used primarily to construct models of how biological brains might function. However, as computers have become increasingly powerful in recent years, the complexity of ANNs has likewise increased such that they are now frequently applied to more practical problems such as:

- 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.

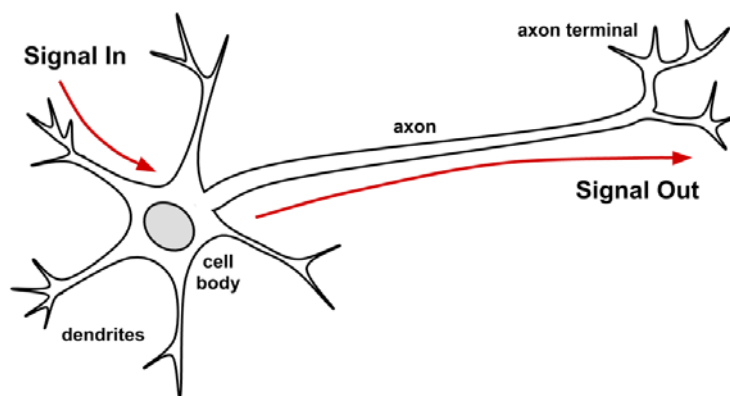


Whether deserving or not, ANN learners are often reported in the media with great fanfare. For instance, an "artificial brain" developed by Google was recently touted for its ability to identify cat videos on YouTube. Such hype may have less to do with anything unique to ANNs and more to do with the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

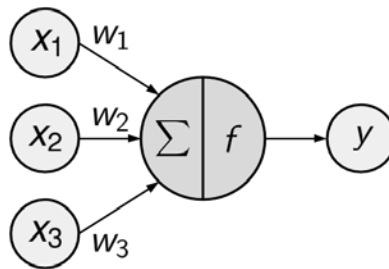
ANNs are best applied to problems where the input data and output data are well-understood or at least fairly simple, yet the process that relates the input to output is extremely complex. As a black box method, they work well for these types of black box problems.

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 that allows the impulse to be weighted according to its relative importance or frequency. As the cell body begins to accumulate the incoming signals, a threshold is reached at which the cell fires and the output signal is then 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, (x), 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 in 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 net input signal 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

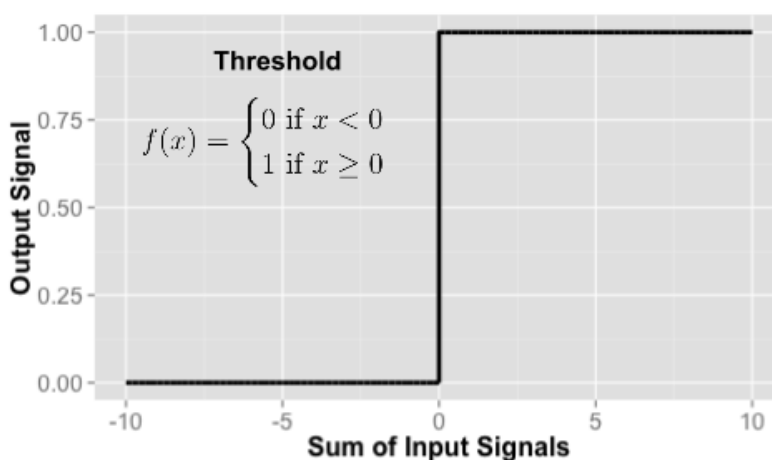
Let's take a look at some of the variations within each of these categories to see how they can be used to construct typical neural network models.

Activation functions

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

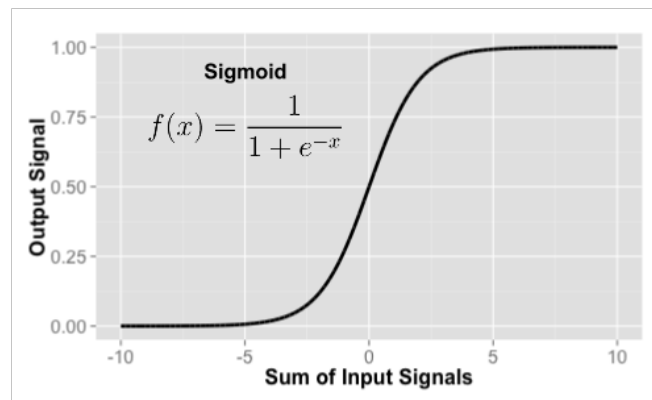
In the biological case, the activation function could be imagined as a process that involves summing the total input signal and determining whether it meets the firing threshold. If so, the neuron passes on the signal; otherwise, it does nothing. In ANN terms, this is known as a **threshold activation 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 of its shape, it is sometimes called a **unit step activation function**.

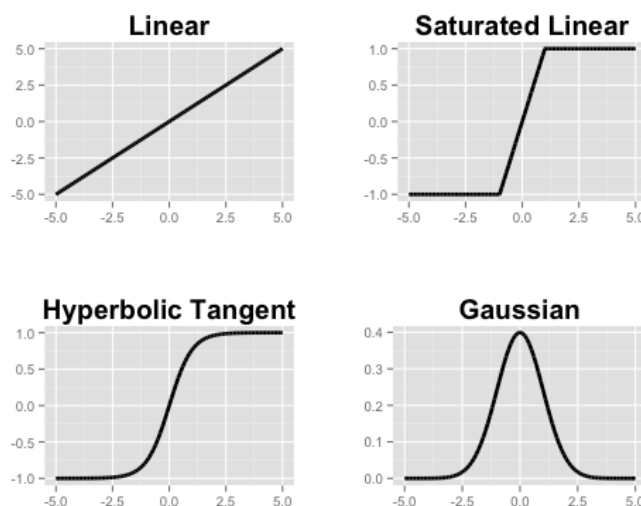


Although the threshold activation function is interesting due to its parallels with biology, it is rarely used in artificial neural networks. Freed from the limitations of biochemistry, ANN activation functions can be chosen based on their ability to demonstrate desirable mathematical characteristics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

Perhaps the most commonly used alternative is the **sigmoid activation function** (specifically the logistic sigmoid) shown in the following figure, where e is the base of natural logarithms (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 for creating efficient ANN optimization algorithms.



Although the 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 as shown:



The primary detail that differentiates among these activation functions is the output signal range. Typically, this is one of $(0, 1)$, $(-1, +1)$, or $(-\infty, +\infty)$. 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**.

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 the sigmoid, the output signal is always 0 or always 1 for an input signal below -5 or above +5, respectively. The compression of the 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 the peaks of sound waves. Because this essentially squeezes the input values into a smaller range of outputs, such 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 feature values fall within a small range around 0. Typically, this is done by standardizing or normalizing the features. By limiting the 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 that you've correctly standardized the input data.

Network topology

The capacity 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:

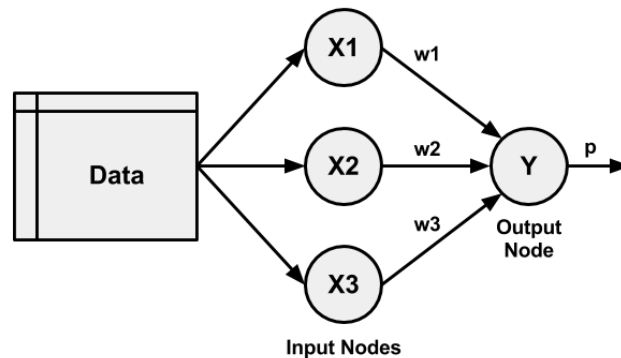
- The number of layers
- Whether information in the network is allowed to travel backward
- The number of nodes within each layer of the network

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.

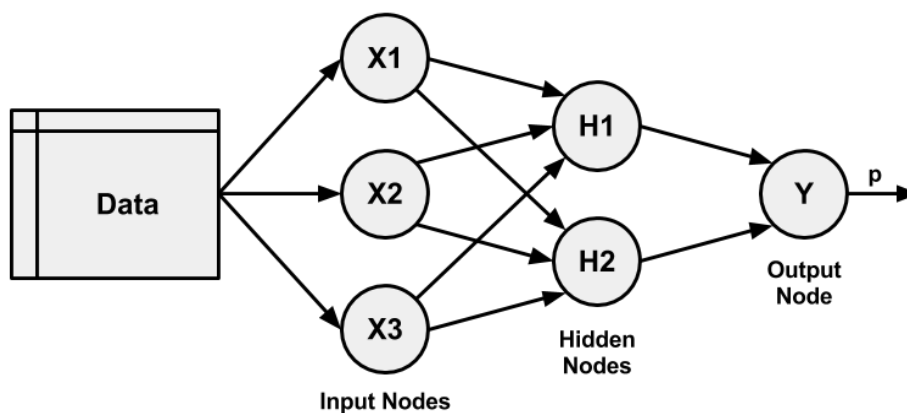
The number of layers

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** receive 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 node's activation function. The signals resulting from 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 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.



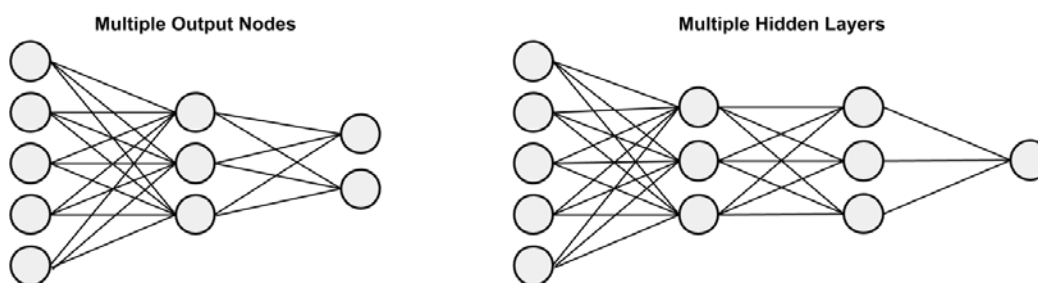
As you might expect, an obvious way to create more complex networks is by adding additional layers. As depicted here, a **multilayer network** adds one or more **hidden layers** that process the signals from the input nodes prior to reaching the output node. Most multilayer networks are **fully connected**,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



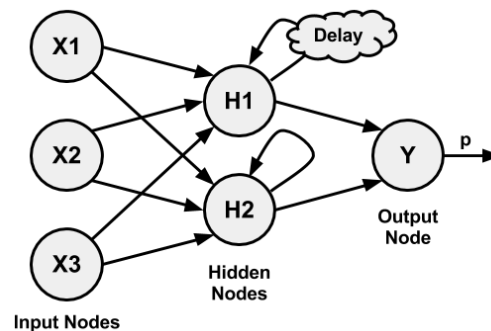
The direction of information travel

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 reaching 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 practice that 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 (labeled **Delay** in the following figure) increases the power of recurrent networks immensely. Notably, this includes the capability to understand 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 shown:



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 without additional clarification, they are most likely referring to a multilayer feedforward network.

The number of nodes in each layer

In addition to 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.

A best practice is to use the fewest nodes that result in adequate performance on 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 sufficiently many neurons is a **universal function approximator**. Essentially, this means that such a network 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 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**.



Interestingly, several research teams of the era independently discovered the backpropagation algorithm. The seminal paper on backpropagation is arguably *Learning representations by back-propagating errors*, *Nature* Vol. 323, pp. 533-566, by D.E. Rumelhart, G.E. Hinton, and R.J. Williams (1986).

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
<ul style="list-style-type: none">• Can be adapted to classification or numeric prediction problems• Among the most accurate modeling approaches• Makes few assumptions about the data's underlying relationships	<ul style="list-style-type: none">• Reputation of being computationally intensive and slow to train, particularly if the network topology is complex• Easy to overfit or underfit 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 iteration of the algorithm is known as an **epoch**. Because the network contains no *a priori* (existing) knowledge, typically the weights are set randomly prior to beginning. Then, the algorithm cycles through the processes until a stopping criterion is reached. The cycles include:

- 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 (or whether) 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, he or she is likely to 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 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.

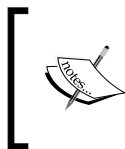
Modeling the strength of concrete with ANNs

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 the use of 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 Yeh's work using a simple neural network model in R.



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

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's 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 non-normal, 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.
0.0000000 0.2663511 0.4000872 0.4171915 0.5457207 1.0000000
```

In comparison, the original minimum and maximum values were 2.33 and 82.6:

```
> summary(concrete$strength)
      Min.    1st Qu.    Median      Mean   3rd Qu.      Max.
2.33000 23.71000 34.44500 35.81796 46.13500 82.60000
```



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 the precedent of *I-Cheng Yeh* 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 that's 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 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:

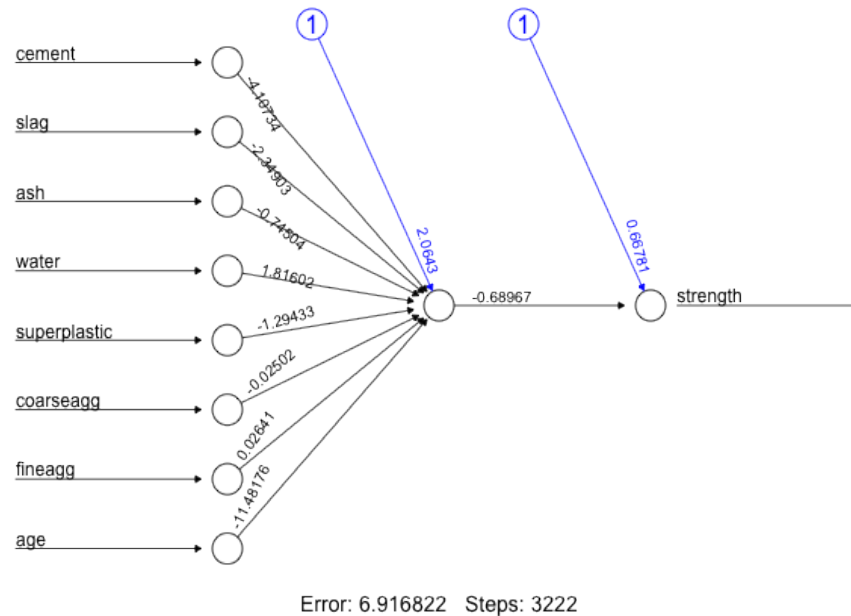
Neural network syntax
using the <code>neuralnet()</code> function in the <code>neuralnet</code> package
<p>Building the model:</p> <pre>m <- neuralnet(target ~ predictors, data = mydata, hidden = 1)</pre> <ul style="list-style-type: none"> • <code>target</code> is the outcome in the <code>mydata</code> data frame to be modeled • <code>predictors</code> is an R formula specifying the features in the <code>mydata</code> data frame to use for prediction • <code>data</code> specifies the data frame in which the <code>target</code> and <code>predictors</code> variables can be found • <code>hidden</code> specifies the number of neurons in the hidden layer (by default, 1) <p>The function will return a neural network object that can be used to make predictions.</p> <p>Making predictions:</p> <pre>p <- compute(m, test)</pre> <ul style="list-style-type: none"> • <code>m</code> is a model trained by the <code>neuralnet()</code> function • <code>test</code> is a data frame containing test data with the same features as the training data used to build the classifier <p>The function will return a list with two components: <code>\$neurons</code>, which stores the neurons for each layer in the network, and <code>\$net.result</code>, which stores the model's predicted values.</p> <p>Example:</p> <pre>concrete_model <- neuralnet(strength ~ cement + slag + ash, data = concrete) model_results <- compute(concrete_model, concrete_data) strength_predictions <- model_results\$net.result</pre>

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 `concrete_model` object:

```
> plot(concrete_model)
```



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 with a 1). The plot also reports the number of training steps and a measure called, the **Sum of Squared Errors (SSE)**. These metrics will be useful when we are evaluating the model performance.

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 our data. To estimate our model's performance, we can use the `compute()` function to generate predictions on the testing dataset:

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

Note that 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.results`, 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 an 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,] 0.7170368646
```

[



Don't be alarmed if your result differs. Because the neural network begins with random weights, the predictions can vary from model to model.

]

Correlations close to 1 indicate strong linear relationships between two variables. Therefore, the correlation here of about 0.72 indicates a fairly strong relationship. This implies that our model is doing a fairly good job, even with only a single hidden node.

[



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. In fact, if you construct a linear model in the same vein as the previous ANN, the correlation is 0.74.

]

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.

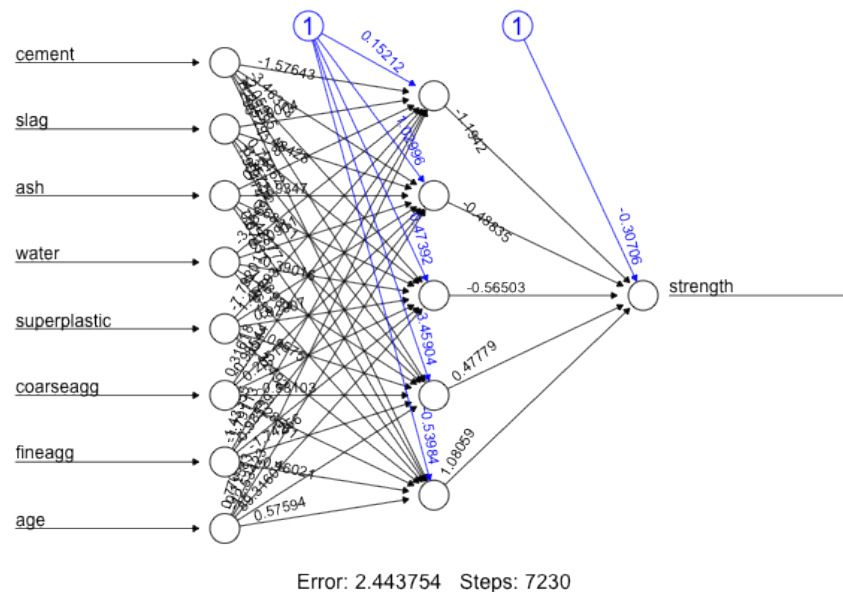
Step 5 – improving model performance

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 parameter `hidden = 5`:

```
> 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. How did this impact performance?

```
> plot(concrete_model2)
```



Notice that the reported error (measured again by SSE) has been reduced from 6.92 in the previous model to 2.44 here. Additionally, the number of training steps rose from 3222 to 7230, which is no surprise given how much more complex the model has become.

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

```
> model_results2 <- compute(concrete_model2, concrete_test[1:8])
> predicted_strength2 <- model_results2$net.result
> cor(predicted_strength2, concrete_test$strength)
      [,1]
[1,] 0.801444583
```

Interestingly, in the original publication, *I-Cheng Yeh* reported a mean correlation of 0.885 using a very similar neural network. For some reason, we fell a bit short. In our defense, he is a civil engineering professor; therefore, he may have applied some subject matter expertise to the data preparation. If you'd like more practice with neural networks, you might try applying the principles learned earlier in this chapter to beat his result, perhaps by 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 defines a boundary between various points of data which represent examples plotted in multidimensional space according to their feature values. The goal of an SVM is to create a flat boundary, called a **hyperplane**, which leads to fairly homogeneous partitions of data on either side. In this way, 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. This has led SVMs to be adopted by a much wider audience who previously might have passed it by due to the somewhat complex math involved with SVM implementation. The good news is that although the math may be difficult, the basic concepts are understandable.

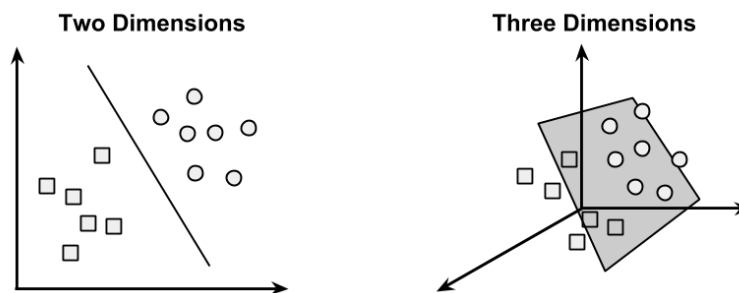
SVMs can be adapted for use with nearly any type of learning task, including both classification and numeric prediction. Many of the algorithm's key successes have come in pattern recognition. Notable applications include:

- 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 organizing 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 when adapting SVMs to other learning tasks such as numeric prediction.

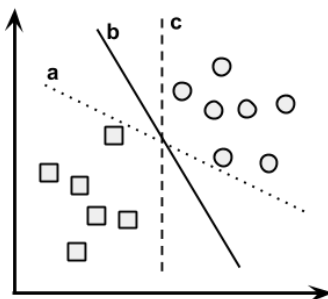
Classification with hyperplanes

As noted previously, SVMs use a linear boundary called a hyperplane to partition data into groups of similar elements, typically as indicated by the class values. For example, the following figure depicts a hyperplane that separates groups of circles and squares in two and three dimensions. Because the circles and squares can be divided 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 data 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.

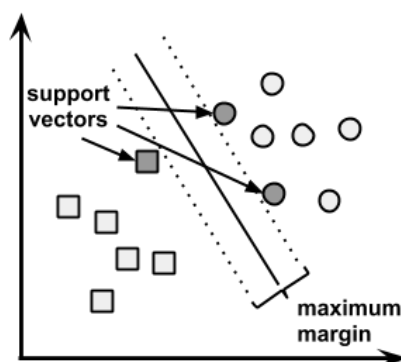
The task of the SVM algorithm is to identify a line that separates the two classes. As shown in the following figure, there is more than one choice of dividing line between the groups of circles and squares. Three such possibilities are labeled **a**, **b**, and **c**. How does the algorithm choose?



Finding the maximum margin

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 future data. This is because slight variations in the positions of the points near the boundary might cause one of them to fall over the line by chance.

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.



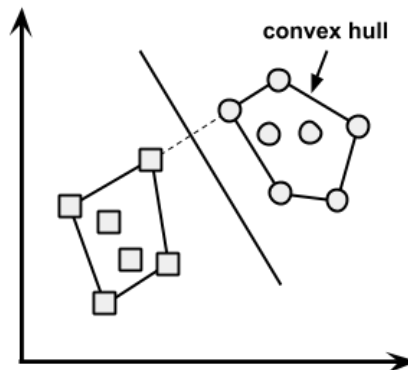
The algorithm to identify the support vectors relies on vector geometry and involves some fairly tricky math which is outside the scope of this book. However, the basic principles of the process are fairly straightforward.



More information on the mathematics of SVMs can be found in the classic paper: *Support-vector network*, *Machine Learning*, Vol. 20, pp. 273-297, by C. Cortes and V. Vapnik (1995). A beginner level discussion can be found in *Support vector machines: hype or hallelujah*, *SIGKDD Explorations*, Vol. 2, No. 2, pp. 1-13, by K.P. Bennett and C. Campbell (2003). A more in-depth look can be found in: *Support Vector Machines* by I. Steinwart and A. Christmann (Springer Publishing Company, 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.

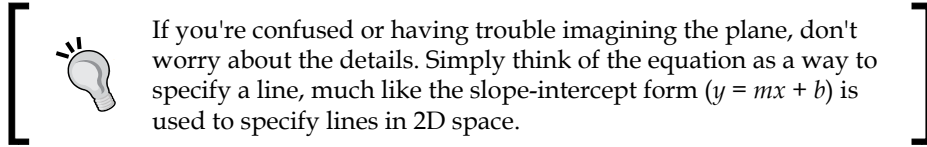


An alternative (but equivalent) approach involves a search through the space of every possible hyperplane in order to find a set of two parallel planes which divide the points into homogeneous groups yet themselves are as far apart as possible. Stated differently, this process is a bit like trying to find the largest mattress that can fit up the 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, $\{w_1, w_2, \dots, w_n\}$, and b is a single number known as the bias.



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). Therefore, in order to maximize distance, we need to minimize $\|w\|$. In order to facilitate finding the solution, the task is typically reexpressed as a set of constraints:

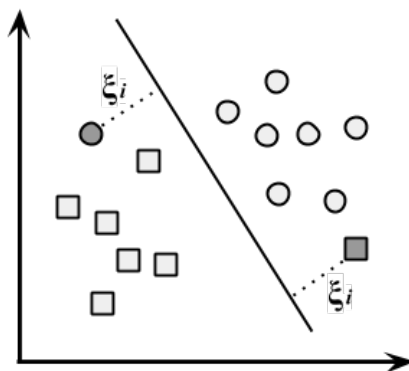
$$\begin{aligned} \min & \frac{1}{2} \|\vec{w}\|^2 \\ \text{s.t. } & y_i (\vec{w} \cdot \vec{x}_i - b) \geq 1, \forall \vec{x}_i \end{aligned}$$

Although this looks messy, it's really not too complicated if you think about it in pieces. Basically, the idea is to minimize the previous formula subject to (s.t.) the condition 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 job 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 non-linearly separable data

As we've worked through the theory behind SVMs, you may be wondering about the elephant in the room: what happens in the case that 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 ξ):



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:

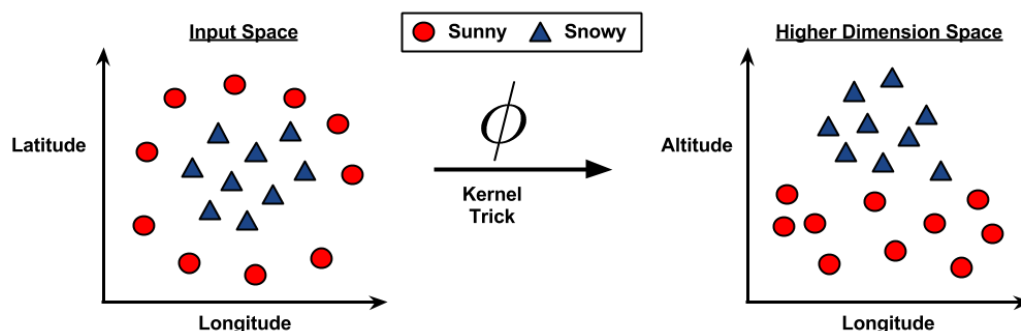
$$\begin{aligned} \min & \frac{1}{2} \|\vec{w}\|^2 + C \sum_{i=1}^n \xi_i \\ \text{s.t. } & y_i (\vec{w} \cdot \vec{x}_i - b) \geq 1 - \xi_i, \forall \vec{x}_i, \xi_i \geq 0 \end{aligned}$$

If you're confused by now, 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 examples that 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 non-linear. 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 non-linearity. 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 non-linear 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 non-linear 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 on the right we are viewing the mountain from ground level. Here, the trend is obvious: snowy weather is found at higher altitudes.

SVMs with non-linear kernels add additional dimensions to the data in order to create separation in this way. Essentially, the kernel trick involves a process of adding 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 the SVM to learn concepts that were not explicitly measured in the original data.

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

Strengths	Weaknesses
<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• 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. Here, the function denoted by the Greek letter phi, that is, $\phi(x)$, is a mapping of the data into another space:

$$K(\vec{x}_i, \vec{x}_j) = \phi(\vec{x}_i) \cdot \phi(\vec{x}_j)$$

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 non-linear 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:

$$K(\vec{x}_i, \vec{x}_j) = \tanh(k \vec{x}_i \cdot \vec{x}_j - \delta)$$

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 for matching 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. That said, in many cases, the choice of kernel is arbitrary, as the performance may vary only slightly. To see how this works in practice, let's apply our understanding of SVM classification to a real-world problem.

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 much matter how it is doing so.

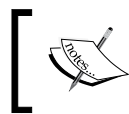
In this section, we will develop a model similar to those used at the core of the **Optical Character Recognition (OCR)** software often bundled with desktop document scanners. The purpose of such software is to process paper-based documents by converting printed or handwritten text into an electronic form to be saved in a database. Of course, this is a difficult problem due to the many variants in handwriting 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 fancy way of 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.

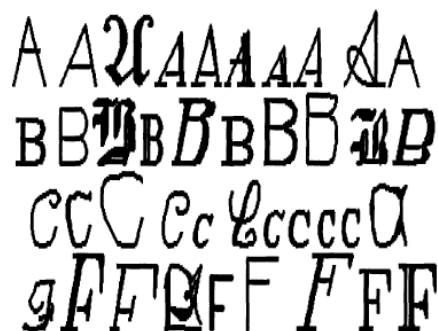
In this exercise, we'll assume that we have already developed the algorithm to partition the document into rectangular regions each consisting of a single character. We will also assume the document contains only alphabetic characters in English. Therefore, we'll simulate a process that involves matching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

Toward 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 about these data, refer to: *Letter recognition using Holland-style adaptive classifiers*, *Machine Learning*, Vol. 6, pp. 161-182, by *W. Frey* and *D.J. Slate* (1991).


The following image, 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's 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 would seem to suggest that we need to normalize or standardize the data. In fact, we can skip this step because the R package that we will use for fitting the SVM model will perform the rescaling for us automatically.

Given that the data preparation has been largely done for us, we can skip directly to the training and testing phases of the machine learning process. In 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) for building the model and the next 4,000 records (20 percent) for testing. 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 from 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 depth in *Chapter 11, Improving Model Performance*).



For a more thorough introduction to `kernlab`, please refer to the author's 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 tangentsigmoid), 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 command `install.packages("kernlab")`. 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 default `type = "response"` was used. This returns a vector containing a predicted letter for each row of values in the testing 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
```

In order 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 also time consuming. We can simplify our evaluation by instead 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 that 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")
```

From there, we make predictions as before:

```
> 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
   281   3719
> prop.table(table(agreement_rbf))
agreement_rbf
  FALSE   TRUE
0.07025 0.92975
```

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 the 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 realize 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. Yet 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.