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 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 how:

- Neural networks mimic living brains to model mathematic 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 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 like a brain uses a network of interconnected cells called **neurons** to provide vast learning capability, the ANN uses a network of artificial neurons or **nodes** to solve challenging 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 in the near future—even a fruit fly with 100,000 neurons far exceeds a state-of-the-art ANN.

Though it may be infeasible 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? This question is the basis of the controversial **Turing test**, 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.

For more about the intrigue and controversy that surrounds the Turing test, refer to the *Stanford Encyclopedia of Philosophy*: `https://plato.stanford.edu/entries/turing-test/`.

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 recent years, the complexity of ANNs has likewise increased so much that they are now frequently applied to more practical problems, including:

- Speech, handwriting, and image recognition programs like those used by smartphone applications, mail sorting machines, and search engines
- The automation of smart devices, such as an office building's environmental controls, or the control of 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 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 often applied to problems where the input data and output data are well-defined, yet the process that relates the input to the output is extremely complex and hard to define. As a black box method, ANNs 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 to accumulate 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**.

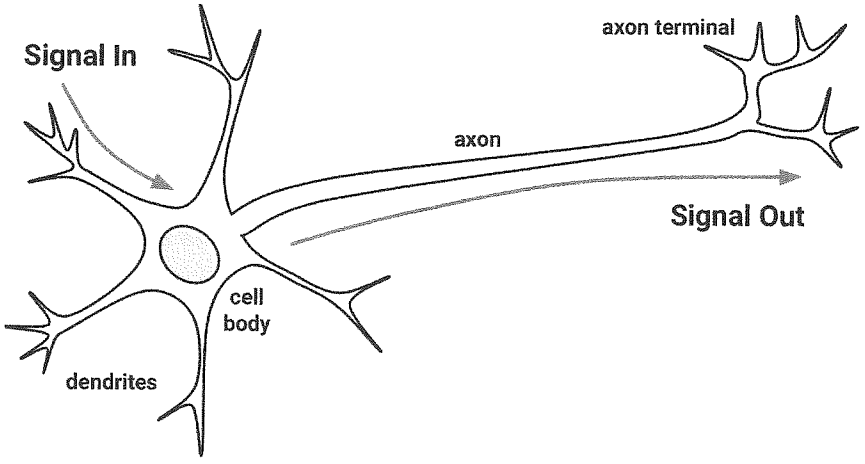


Figure 7.1: An artistic depiction of a biological neuron

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.

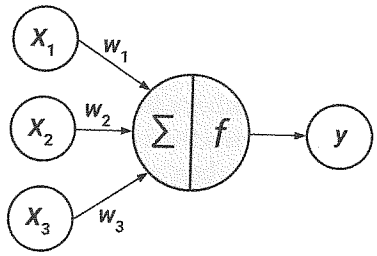


Figure 7.2: An artificial neuron is designed to mimic the structure and function of a biological neuron

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 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**, which 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 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 its shape resembles a stair, it is sometimes called a **unit step activation function**.

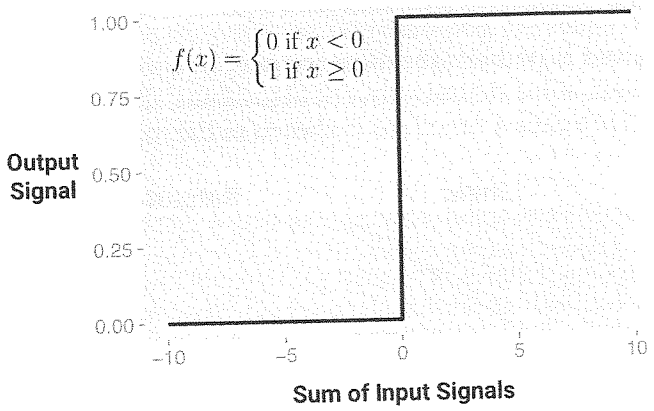


Figure 7.3: The threshold activation function is "on" only after the input signals meet a threshold

Although the threshold activation function is interesting due to its parallels with biology, it is rarely used in ANNs. Freed from the limitations of biochemistry, ANN activation functions can be chosen based on their ability to demonstrate desirable mathematical characteristics and their ability to 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 zero to one.

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.

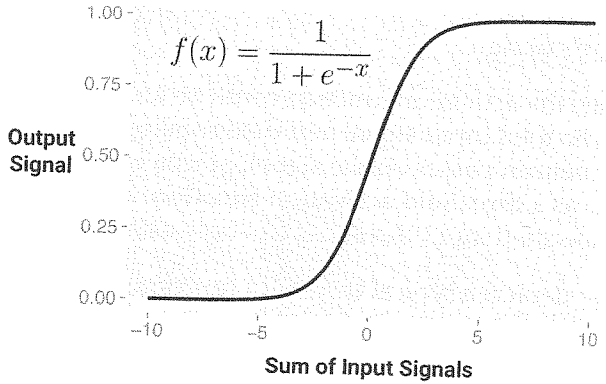


Figure 7.4: The sigmoid activation function mimics the biological activation function with a smooth curve

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 shown in the following figure:

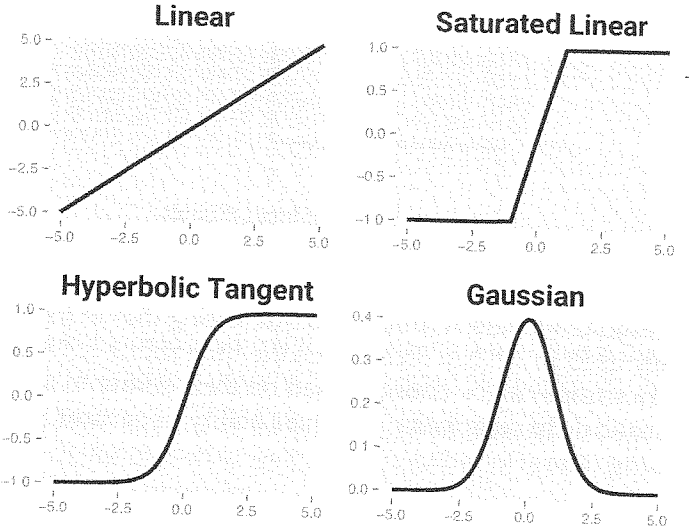


Figure 7.5: Several common neural network activation functions

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 is the basis of 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 the sigmoid, the output signal is very near zero for an input signal below negative five and very near one for an input signal above positive five. 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 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**.

One solution to the squashing problem is to transform all neural network inputs such that the feature values fall within a small range around zero. This may involve standardizing or normalizing the features. By restricting the range of input values, the activation function will have action across the entire range. 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 failing to converge, double-check that you've correctly standardized the input data. Choosing a different activation function may also be appropriate.

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 more 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 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 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.

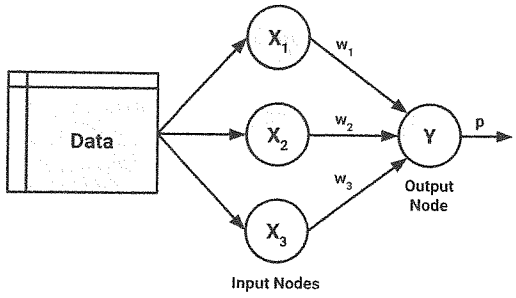


Figure 7.6: A simple single-layer ANN with three input nodes

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.

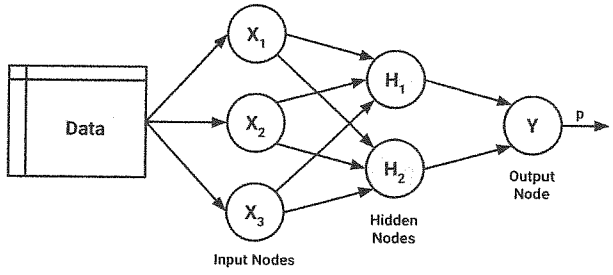


Figure 7.7: A multilayer network with a single two-node hidden layer

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 the input layer to 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 networks is referred to as **deep learning**. Deep neural networks trained on large datasets are capable of human-like performance on complex tasks like image recognition and text processing.

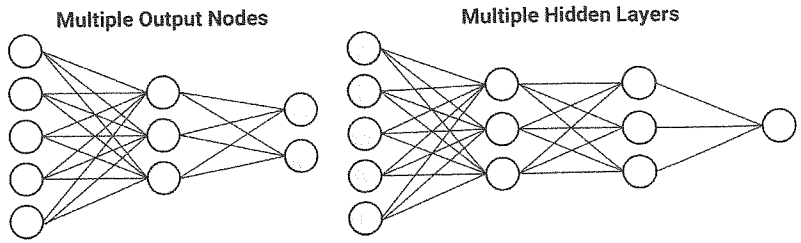


Figure 7.8: Complex ANNs can have multiple output nodes or multiple hidden layers

In contrast to feedforward networks, a **recurrent network** (or **feedback network**) allows signals to travel backward 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 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:

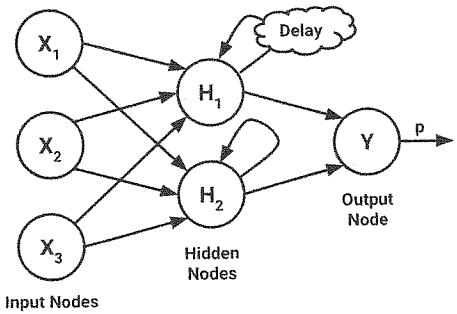


Figure 7.9: Allowing information to travel backward in the network can model a time delay

DNNs and recurrent networks are increasingly being used for a variety of high-profile applications and consequently have become highly popular. However, building such networks uses techniques and software outside the scope of this book, and often requires access to specialized computing hardware or cloud servers. On the other hand, simpler feedforward networks are also very capable of modeling many real-world tasks. In fact, the multilayer feedforward network, also known as the **multilayer perceptron (MLP)**, is the de facto standard ANN topology. If you are interested in deep learning, understanding the MLP topology provides a strong theoretical basis for building more complex DNN models later on.

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.

The 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**. 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 *Learning representations by back-propagating errors, Rumelhart, DE, Hinton, GE, Williams, RJ, Nature, 1986, Vol. 323, pp. 533-566.*

Although still somewhat computationally expensive 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 algorithm 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, this 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 training time at the risk of overshooting the valley.

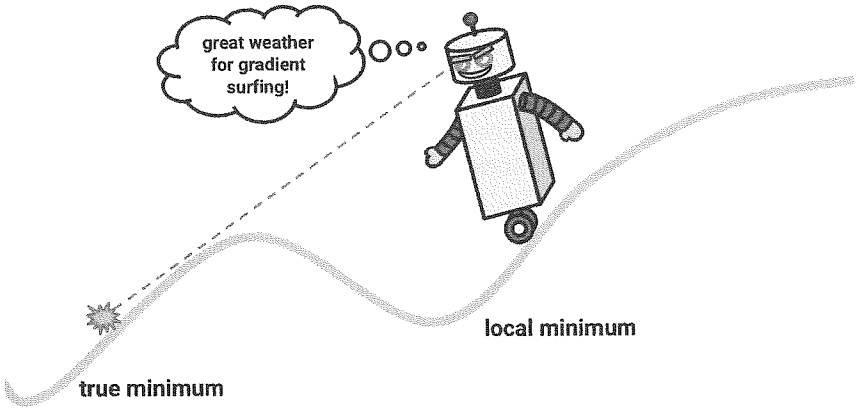


Figure 7.10: The gradient decent algorithm seeks the minimum error but may also find a local minimum

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 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, Yeh, IC, Cement and Concrete Research, 1998, Vol. 28, pp. 1797-1808.*

According to the website, the 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 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 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 zero to one 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 zero and one, 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.

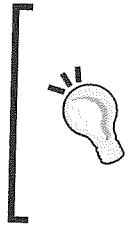
Following Yeh's precedent in the original publication, we will partition the data into a training set with 75 percent of the examples and a testing set with 25 percent. The CSV file we used was already sorted in random order, so we simply need to divide it into two portions:

```
> 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 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 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, act.fct = "logistic")
```

- `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 where the `target` and `predictors` are found
- `hidden` specifies the number of neurons in the hidden layer (by default, 1)
  note: use an integer vector to specify multiple hidden layers, e.g., `c(2, 2)`
- `act.fct` specifies the activation function, either `"logistic"` or `"tanh"`
  note: a *differentiable* custom activation function can also be supplied

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, hidden = c(5, 5), act.fct = "tanh")
model_results <- compute(concrete_model, concrete_data)
strength_predictions <- model_results$net.result
```

We'll begin by training the simplest multilayer feedforward network with the default settings using 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)
```

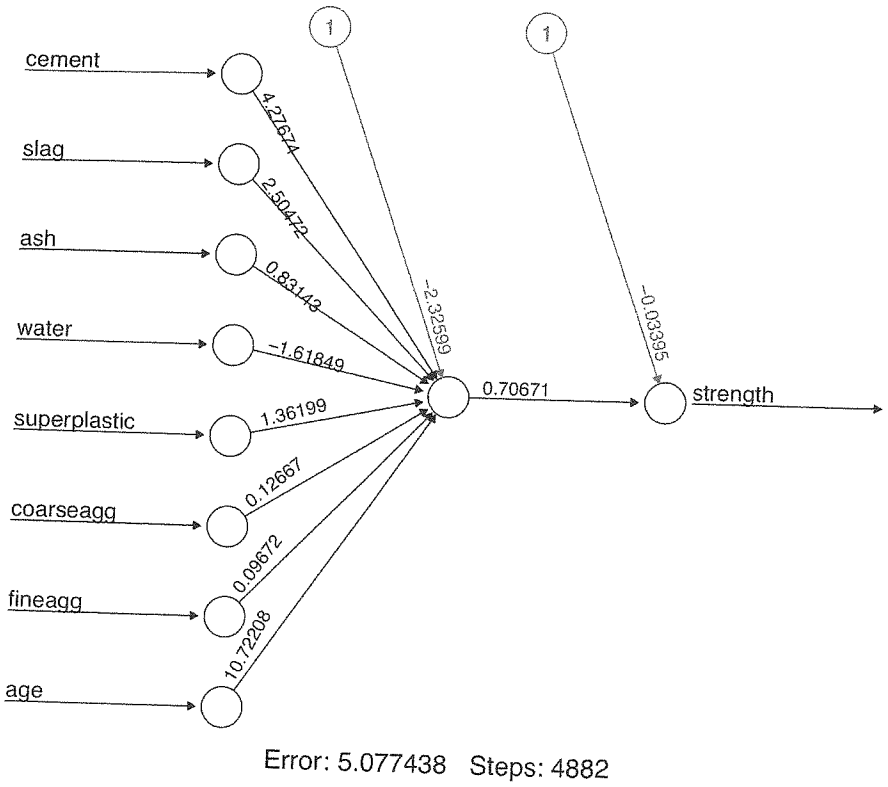


Figure 7.11: Topology visualization of our simple multilayer feedforward network

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 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 beta 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 differences between the predicted and actual values. The lower the SSE, the more closely the model conforms to the training data, which tells us about performance on the training data but 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()` function 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'll measure the correlation between our predicted concrete strength and the true value. If the predicted and actual values are highly correlated, the model is likely to be a useful gauge of concrete strength.

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 one 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 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)
```

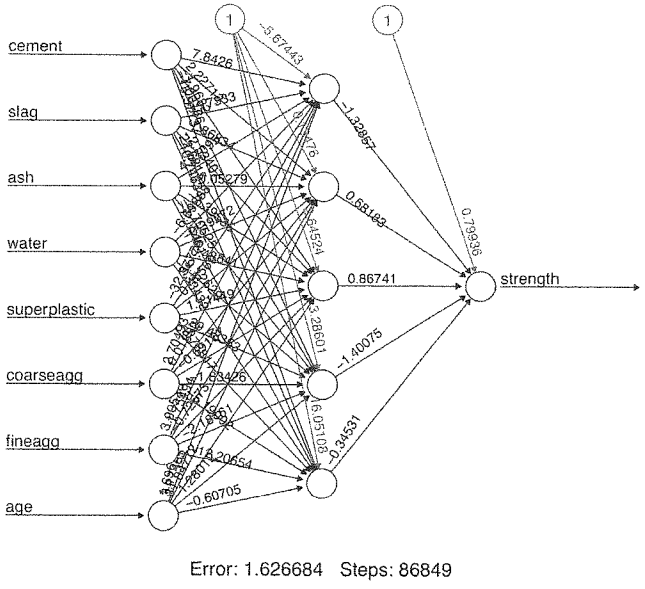


Figure 7.12: Topology visualization of an increased number of hidden nodes

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
```

Despite these substantial improvements, there is still more we can do to attempt to improve the model performance. In particular, we have the ability to add additional hidden layers and to change the network's activation function. In making these changes, we create the foundations of a very simple deep neural network.

The choice of activation function is usually very important for deep learning. The best function for a particular learning task is typically identified through experimentation, then shared more widely within the machine learning research community.

Recently, an activation function known as a **rectifier** has become extremely popular due to its success on complex tasks such as image recognition. A node in a neural network that uses the rectifier activation function is known as a **rectified linear unit** (**ReLU**). As depicted in the following figure, the rectifier activation function is defined such that it returns x if x is at least zero, and zero otherwise. The significance of this function is due to the fact that it is nonlinear yet has simple mathematical properties that make it both computationally inexpensive and highly efficient for gradient descent. Unfortunately, its derivative is undefined at $x = 0$ and therefore cannot be used with the `neuralnet()` function.

Instead, we can use a smooth approximation of the ReLU known as **softplus** or **SmoothReLU**, an activation function defined as $log(1 + e^x)$. As shown in the following figure, the softplus function is nearly zero for x less than zero and approximately x when x is greater than zero:

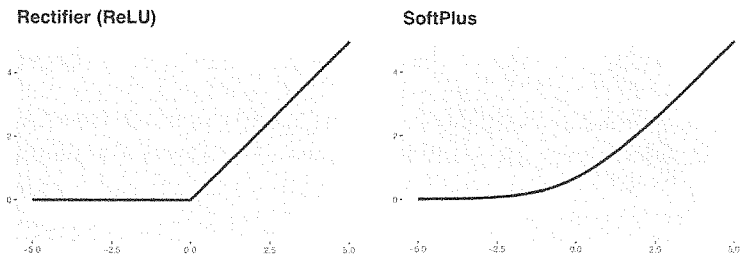


Figure 7.13: The softplus activation function provides a smooth, differentiable approximation of ReLU

To define a `softplus()` function in R, use the following code:

```
> softplus <- function(x) { log(1 + exp(x)) }
```

This activation function can be provided to `neuralnet()` using the `act.fct` parameter. Additionally, we will add a second hidden layer of five nodes by supplying the `hidden` parameter the integer vector `c(5, 5)`. This creates a two-layer network, each having five nodes, all using the softplus activation function:

```
> set.seed(12345)
> concrete_model3 <- neuralnet(strength ~ cement + slag +
                               ash + water + superplastic +
                               coarseagg + fineagg + age,
                               data = concrete_train,
                               hidden = c(5, 5),
                               act.fct = softplus)
```

As before, the network can be visualized:

```
> plot(concrete_model3)
```

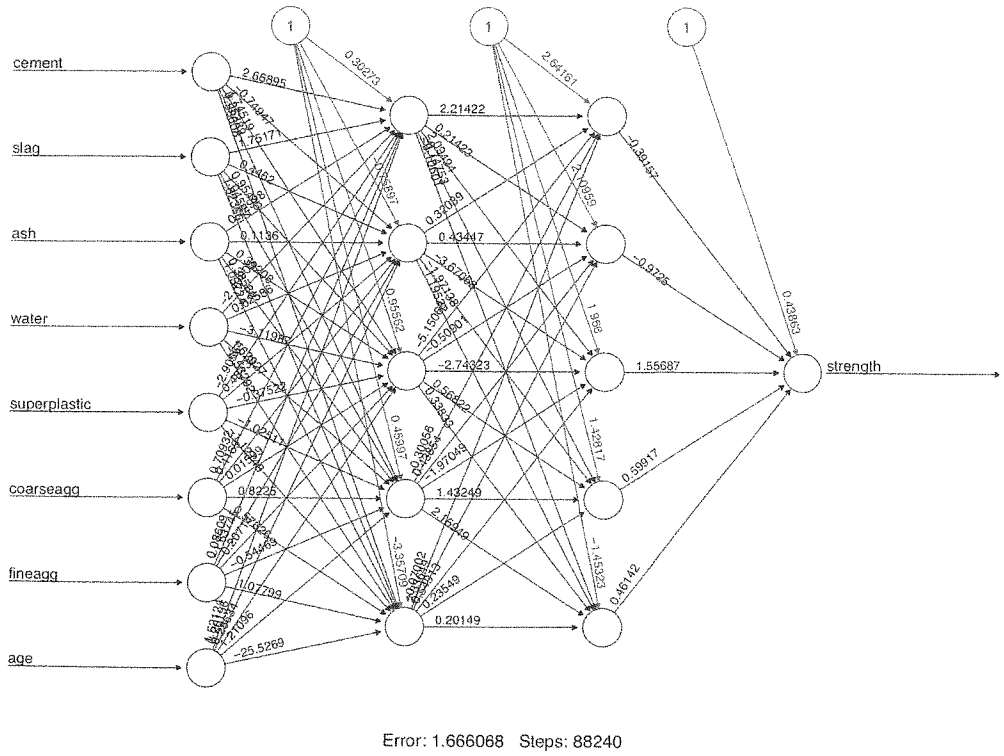


Figure 7.14: Visualizing our network with two layers of hidden nodes using the softplus activation function

And the correlation between the predicted and actual concrete strength can be computed:

```
> model_results3 <- compute(concrete_model3, concrete_test[1:8])
> predicted_strength3 <- model_results3$net.result
> cor(predicted_strength3, concrete_test$strength)
             [,1]
[1,] 0.9348395359
```

The correlation between the predicted and actual strength was 0.935, which is our best performance yet. Interestingly, in the original publication, Yeh reported a correlation of 0.885. This means that with relatively little effort, we were able to match and even exceed the performance of a subject matter expert. Of course, Yeh's results were published in 1998, giving us the benefit of over 20 years of additional neural network research!

One important thing to be aware of is that, because we had normalized the data prior to training the model, the predictions are also on a normalized scale from zero to one. For example, the following code shows a data frame comparing the original dataset's concrete strength values to their corresponding predictions side-by-side:

```
> strengths <- data.frame(
    actual = concrete$strength[774:1030],
    pred = predicted_strength3
  )
> head(strengths, n = 3)
    actual         pred
774  30.14 0.2860639091
775  44.40 0.4777304648
776  24.50 0.2840964250
```

Examining the correlation, we see that the choice of normalized or unnormalized data does not affect our computed performance statistic—the correlation of 0.935 is exactly the same as before:

```
> cor(strengths$pred, strengths$actual)
[1] 0.9348395359
```

However, if we were to compute a different performance metric, such as the absolute difference between the predicted and actual values, the choice of scale would matter quite a bit.

With this in mind, we can create an `unnormalize()` function that reverses the min-max normalization procedure and allow us to convert the normalized predictions to the original scale:

```
> unnormalize <- function(x) {
    return((x * (max(concrete$strength)) -
            min(concrete$strength)) + min(concrete$strength))
  }
```

After applying the custom `unnormalize()` function to the predictions, we can see that the new predictions are on a similar scale to the original concrete strength values. This allows us to compute a meaningful absolute error value. Additionally, the correlation between the unnormalized and original strength values remains the same:

```
> strengths$pred_new <- unnormalize(strengths$pred)
> strengths$error <- strengths$pred_new - strengths$actual

> head(strengths, n = 3)
    actual          pred    pred_new         error
774  30.14 0.2860639091 23.62887889 -6.511121108
775  44.40 0.4777304648 39.46053639 -4.939463608
776  24.50 0.2840964250 23.46636470 -1.033635298

> cor(strengths$pred_new, strengths$actual)
[1] 0.9348395359
```

When applying neural networks to your own projects, you will need to perform a similar series of steps to return the data to its original scale.

You may also find that neural networks quickly become much more complicated as they are applied to more challenging learning tasks. For example, you may find that you run into the so-called **vanishing gradient problem** and closely-related **exploding gradient problem**, where the backpropagation algorithm fails to find a useful solution due to an inability to converge in a reasonable time. As a remedy to these problems, one may perhaps try varying the number of hidden nodes, applying different activation functions such as the ReLU, adjusting the learning rate, and so on. The `?neuralnet` help page provides more information on the various parameters that can be adjusted. This leads to another problem, however, in which testing a large number of parameters becomes a bottleneck to building a strong-performing model. This is the tradeoff of ANNs and even more so, DNNs: harnessing their great potential requires a great investment of time and computing power.

Just as is often the case in life more generally, it is possible to trade time and money in machine learning. Using paid cloud computing resources such as Amazon Web Services (AWS) and Microsoft Azure allows one to build more complex models or test many models more quickly. More on this subject can be found in *Chapter 12, Specialized Machine Learning Topics.*

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 a multidimensional space representing examples and their feature values. The goal of an 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, 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, interest in them grew greatly after they were adopted by the machine learning community. Their popularity exploded after high-profile success stories on difficult learning problems, as well as the development of award-winning SVM algorithms that were implemented in well-supported libraries across many programming languages, including R. SVMs have thus been adopted by a wide audience, which might have otherwise been unable to apply the somewhat complex mathematics needed to implement an SVM. The good news is that although the mathematics 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 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. Principles similar to those presented here also apply when adapting SVMs for 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 a 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 were the points are not linearly separable.

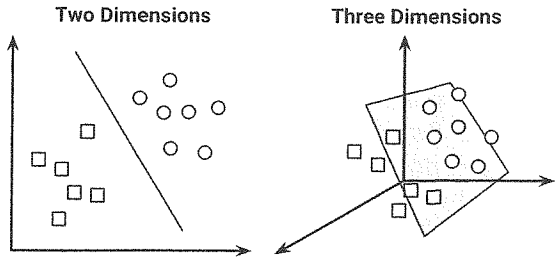


Figure 7.15: The squares and circles are linearly separable in both two and three dimensions

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?

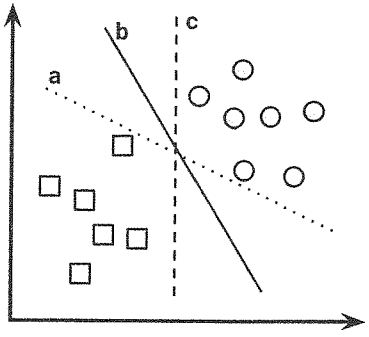


Figure 7.16: Three of many potential lines dividing the squares and circles

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 of the data points, the line that leads to the greatest separation will generalize the best to future data. The maximum margin will improve the chance that even if random noise is added, each class will remain on its own 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. The support vectors alone 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.

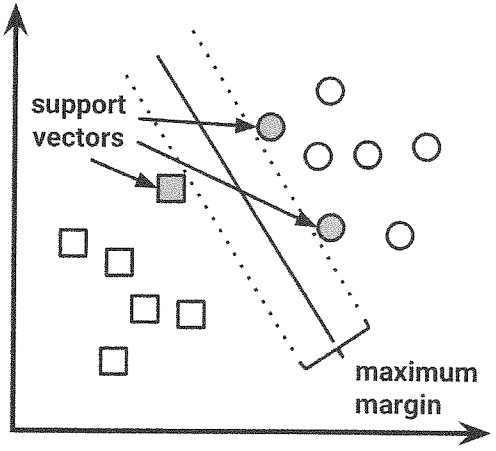


Figure 7.17: The maximum margin hyperplane is defined by the support vectors

The algorithm to identify the support vectors relies on vector geometry and involves some fairly tricky mathematics 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 *Support-Vector Networks, Cortes, C* and *Vapnik, V, Machine Learning, 1995, Vol. 20, pp. 273-297.* A beginner-level discussion can be found in *Support Vector Machines: Hype or Hallelujah?, Bennett, KP* and *Campbell, C, SIGKDD Explorations, 2000, Vol. 2, pp. 1-13.* A more in-depth look can be found in *Support Vector Machines, Steinwart, I* and *Christmann, A, New York: Springer, 2008.*

The case of linearly separable data

Finding the maximum margin is easiest 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.

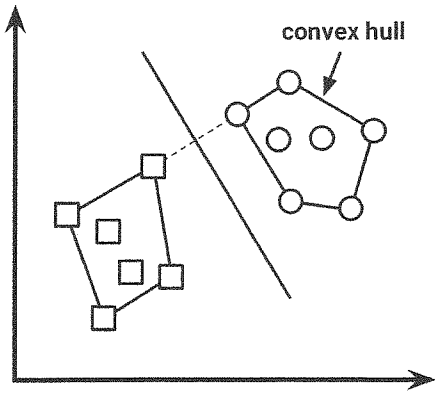


Figure 7.18: The MMH is the perpendicular bisector of the shortest path between convex hulls

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 in multidimensional space, don't worry about the details. Simply think of the equation as a way to specify a surface, much like 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 the denominator, to maximize distance we need to minimize $\|w\|$. The task is typically re-expressed as a set of constraints, as follows:

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

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 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 Xi):

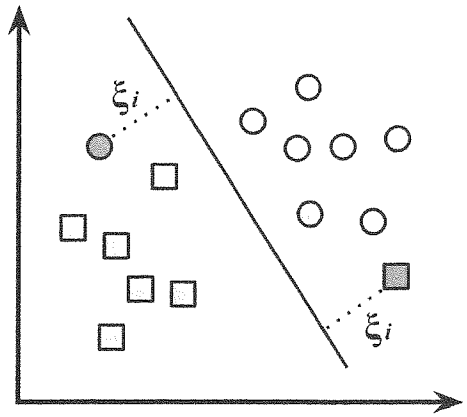


Figure 7.19: Points falling on the wrong side of the boundary come with a cost penalty

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 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 points 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 nonlinear spaces

In many real-world datasets, the relationships between variables are nonlinear. As we just discovered, an 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.

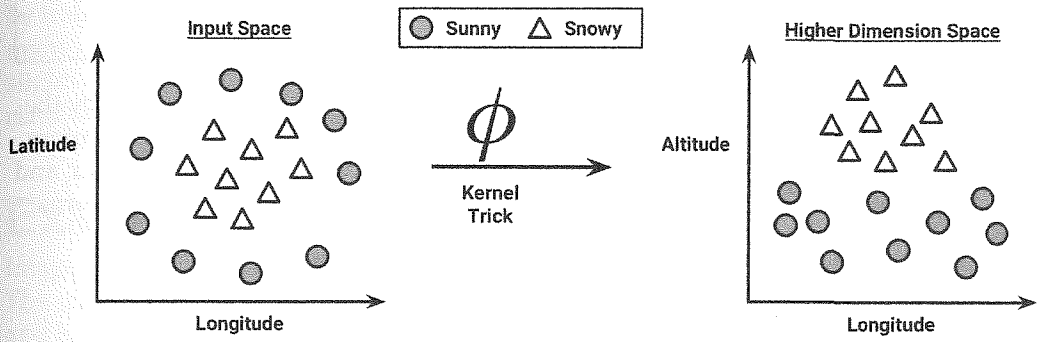


Figure 7.20: The kernel trick can help transform a nonlinear problem into a linear one

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 a distance at ground level. Here, the trend is obvious: snowy weather is found at higher altitudes.

In this way, SVMs with nonlinear kernels add additional dimensions to the data in order to create separation. 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 the 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 • Gained popularity due to their high accuracy and high-profile wins in data mining competitions	• Finding the best model requires the 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, $\phi(x)$, 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. 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 an SVM model somewhat analogous to a neural network using a sigmoid activation function. The Greek letters kappa and delta are used as kernel parameters:

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

The **Gaussian RBF kernel** is similar to an 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:

$$K\left(\vec{x}_i, \vec{x}_j\right) = e^{\frac{-\left\|\vec{x}_i - \vec{x}_j\right\|^2}{2\sigma^2}}$$

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.

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 or in smartphone applications. 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 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 can be combined 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 glyph. 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 to Z.

To this end, we'll use a dataset donated to the UCI Machine Learning 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, Slate, DJ* and *Frey, PW, Machine Learning, 1991, Vol. 6, pp. 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:

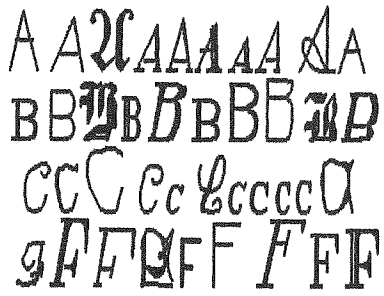


Figure 7.21: Examples of glyphs the SVM algorithm will attempt to identify

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, it 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 ...
```

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 there is no data preparation left to perform, we can move 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, see `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 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
```

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 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 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

Let's take a moment to contextualize the performance of the SVM model we trained to identify letters of the alphabet from image data. With one line of R code, the model was able to achieve an accuracy of nearly 84 percent, which slightly surpassed the benchmark percent published by academic researchers in 1991. Although an accuracy of 84 percent is not nearly high enough to be useful for OCR software, the fact that a relatively simple model can reach this level is a remarkable accomplishment in itself. Keep in mind that the probability the model's prediction would match the actual value by dumb luck alone is quite small at under four percent. This implies that our model performs over 20 times better than random chance. As remarkable as this is, perhaps by adjusting the SVM function parameters to train a slightly more complex model, we can also find that the model is useful in the real world.

To calculate the probability of the SVM model's predictions matching the actual values by chance alone, apply the joint probability rule for independent events covered in *Chapter 4, Probabilistic Learning – Classification Using Naive Bayes*. Because there are 26 letters, each appearing at approximately the same rate in the test set, the chance that any one letter is predicted correctly is *(1 / 26) * (1 / 26)*. Since there are 26 different letters, the total probability of agreement is *26 * (1 / 26) * (1 / 26) = 0.0384*, or 3.84 percent.

Changing the SVM kernel function

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 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
  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 `RNGversion("3.5.2")` and `set.seed(12345)` prior to running the `ksvm()` function.

Simply by changing the kernel function, we were able to increase the accuracy of our character recognition model from 84 percent to 93 percent.

Identifying the best SVM cost parameter

If this level of performance is still unsatisfactory for the OCR program, it is certainly possible to test additional kernels. However, another fruitful approach is to vary the cost parameter, which modifies the width of the SVM decision boundary. This governs the model's balance between overfitting and underfitting the training data—the larger the cost value, the harder the learner will try to perfectly classify every training instance, as there is a higher penalty for each mistake. On the one hand, a high cost can lead the learner to overfit the training data. On the other hand, a cost parameter set too small can cause the learner to miss important, subtle patterns in the training data and underfit the true pattern.

There is no rule of thumb to know the ideal value beforehand, so instead we will examine how the model performs for various values of `C`, the cost parameter. Rather than repeating the training and evaluation process repeatedly, we can use the `sapply()` function to apply a custom function to a vector of potential cost values. We begin by using the `seq()` function to generate this vector as a sequence counting from five to 40 by five. Then, as shown in the following code, the custom function trains the model as before, each time using the cost value and making predictions on the test dataset. Each model's accuracy is computed as the number of predictions that match the actual values divided by the total number of predictions. The result is visualized using the `plot()` function:

```
> cost_values <- c(1, seq(from = 5, to = 40, by = 5))
>
> accuracy_values <- sapply(cost_values, function(x) {
    set.seed(12345)
    m <- ksvm(letter ~ ., data = letters_train,
              kernel = "rbfdot", C = x)
    pred <- predict(m, letters_test)
    agree <- ifelse(pred == letters_test$letter, 1, 0)
    accuracy <- sum(agree) / nrow(letters_test)
    return (accuracy)
  })

> plot(cost_values, accuracy_values, type = "b")
```

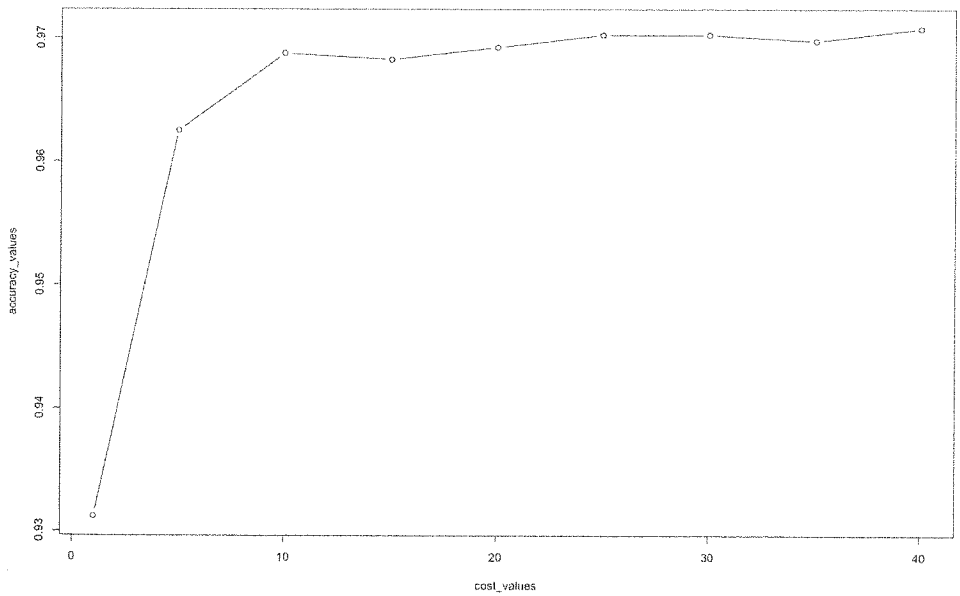


Figure 7.22: Mapping accuracy against SVM cost for the RBF kernel

As depicted in the visualization, with an accuracy of 93 percent, the default SVM cost parameter of `C = 1` resulted in by far the least accurate model among the nine models evaluated. Instead, setting `C` to a value of 10 or higher results in an accuracy of around 97 percent, which is quite an improvement in performance! Perhaps this is close enough to perfect for the model to be deployed in a real-world environment, though it may still be worth experimenting further with various kernels to see if it is possible to get even closer to 100 percent accuracy. Each additional improvement in accuracy will result in fewer mistakes for the OCR software and a better overall experience for the end user.

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 that you learned here, you should be capable of picking up the nuances that distinguish the many advancements that are being developed every day, including the ever-growing field of deep learning.

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.