

# 10

## Evaluating Model Performance

Many years ago, when only the wealthy could afford access to education, tests and examinations were not used to evaluate the students. Instead, they were used to judge the teachers—parents wanted to know whether their children were learning enough to justify the instructors' wages. Obviously, this practice has changed over the years. Now, such evaluations are used to distinguish between high and low-achieving students, filtering them into careers and further educational opportunities.

Given the significance of this process, a great deal of effort is invested in developing accurate student assessments. A fair assessment will have a large number of questions to cover a wide breadth of topics and reward true knowledge over lucky guesses. The assessment should also include some questions requiring the student to think about a problem he or she has never faced before. Correct responses would indicate that the student can apply the knowledge more generally.

A similar process of exam writing can be used to imagine the practice of evaluating machine learners. As different algorithms have varying strengths and weaknesses, it is necessary to use tests that reveal distinctions among the learners when measuring how a learner will perform on future data.

This chapter provides the information needed to assess machine learners, such as:

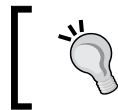
- The reasons why predictive accuracy is not sufficient to measure performance, and the performance measures you might use instead
- Methods to ensure that the performance measures reasonably reflect a model's ability to predict or forecast unseen data
- How to use R to apply these more useful measures and methods to the predictive models we learned in previous chapters

As you will discover, just as the best way to learn a topic is to attempt to teach it to someone else, the process of teaching machine learners will also provide you with a greater insight into how to better the use of machine learning methods you've learned so far.

## Measuring performance for classification

To measure classification performance in previous chapters, we used a measure of accuracy that divided the proportion of correct predictions by the total number of predictions. This number indicates the percentage of cases in which the learner is right or wrong. For instance, suppose a classifier correctly identified whether or not 99,990 out of 100,000 newborn babies are carriers of a treatable but potentially-fatal genetic defect. This would imply an accuracy of 99.99 percent and an error rate of only 0.01 percent.

Although this would appear to indicate an extremely accurate classifier, it would be wise to collect additional information before trusting your child's life to the test. What if the genetic defect is found in only 10 out of every 100,000 babies? A test that predicts "no defect" regardless of circumstances will still be correct for 99.99 percent of all cases. In this case, even though the predictions are correct for the large majority of data, the classifier is not very useful for its intended purpose, which is to identify children with birth defects.



This is one consequence of the **class imbalance problem**, which refers to the trouble associated with data having a large majority of records belonging to a single class.

The best measure of classifier performance is whether the classifier is successful at its intended purpose. For this reason, it is crucial to have measures of model performance that measure utility rather than raw accuracy. Toward this end, we will begin working with a variety of measures derived from predictions presented in a familiar format: the confusion matrix. Before we get started, however, we need to consider how to prepare classification results for evaluation.

## Working with classification prediction data in R

There are three main types of data that are used to evaluate a classifier:

- Actual class values
- Predicted class values
- Estimated probability of the prediction

We used the first two types in previous chapters. The idea is to maintain two vectors of data: one holding the true, or actual class values and the other holding the predicted class values. Both vectors must have the same number of values stored in the same order. The predicted and actual values may be stored as separate R vectors or columns in a single R data frame. Either of these approaches will work with most R functions.

The actual class values come directly from the target feature in the test dataset. For instance, if your test data are in a data frame named `test_data`, and the target is in a column named `outcome`, we can create a vector of actual values using a command similar to `actual_outcome <- test_data$outcome`.

Predicted class values are obtained using the model. For most machine learning packages, this involves applying the `predict()` function to a model object and a data frame of test data, such as: `predicted_outcome <- predict(model, test_data)`.

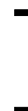
Until now, we have only examined classification predictions using these two vectors of data. Yet hidden behind-the-scenes is another piece of useful information. Even though the classifier makes a single prediction about each example, it may be more confident about some decisions than others. For instance, a classifier may be 99 percent certain that a SMS with the words "free" and "ringtones" is 99 percent spam, but is only 51 percent certain that a SMS with the word "tonight" is spam. In both cases, the classifier predicts a spam, but it is far more certain about one decision than the other.

Studying these internal prediction probabilities is useful to evaluate the model performance and is the source of the third type of evaluation data. If two models make the same number of mistakes, but one is more able to accurately assess its uncertainty, then it is a smarter model. It's ideal to find a learner that is extremely confident in making a correct prediction, but timid in the face of doubt. The balance between confidence and caution is a key part of model evaluation.

Unfortunately, obtaining the internal prediction probabilities can be tricky because the method for doing so varies across classifiers. In general, the `predict()` function for the classifier will allow you to specify the type of prediction you want. To obtain a single predicted class, such as spam or ham, you typically specify "class" type. To obtain the prediction probability, you typically specify a type such as `prob`, `posterior`, `raw`, or `probability`.



Nearly all of the classifiers presented in this book will provide such probabilities; the parameter for doing so is included in the syntax box introducing each model.



For example, to output the predicted probabilities for a naive Bayes classifier as described in *Chapter 4, Probabilistic Learning – Classification Using Naive Bayes*, you would use `type = "raw"` with the prediction function, such as: `predicted_prob <- predict(model, test_data, type = "raw")`.


Similarly, the command for a C5.0 classifier as described in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules* is: `predicted_prob <- predict(model, test_data, type = "prob")`.

Keep in mind that in most cases the `predict()` function will return a probability for each level of the outcome. For example, in the case of a two-outcome yes/no model, the `predicted_prob` might be a matrix or data frame as shown in the following expression:

```
> head(predicted_prob)
      no      yes
1 0.0808272 0.9191728
2 1.0000000 0.0000000
3 0.7064238 0.2935762
4 0.1962657 0.8037343
5 0.8249874 0.1750126
6 1.0000000 0.0000000
```

Be careful while constructing an evaluation dataset to ensure that you are using the correct probability for the class level of interest. To avoid confusion, in the case of a binary outcome, you might even consider dropping the vector for one of the two alternatives.

To illustrate typical evaluation data, we'll use a data frame containing predicted class values, actual class values, and the estimated probability of a spam as determined by the SMS spam classification model developed in *Chapter 4, Probabilistic Learning: Classification Using Naive Bayes*.

 To follow along with the examples here, download the `sms_results.csv` file from the Packt Publishing's website and load to a data frame using the command:  
`sms_results <- read.csv("sms_results.csv")`

The `sms_results` data frame is simple; shown in the following command and its output, it contains three vectors of 1,390 values. One vector contains values indicating the actual type of SMS message (`spam` or `ham`), one vector indicates the model's predicted type, and the third vector indicates the probability that the message was spam:

```
> head(sms_results)
  actual_type predict_type   prob_spam
1        ham          ham 2.560231e-07
2        ham          ham 1.309835e-04
3        ham          ham 8.089713e-05
4        ham          ham 1.396505e-04
5        spam          spam 1.000000e+00
6        ham          ham 3.504181e-03
```

Notice that when the predicted type is `ham`, the `prob_spam` value is extremely close to zero. Conversely, when the predicted type was `spam`, the `prob_spam` value is equal to one, which implies that the model was 100 percent certain that the SMS was spam. The fact that the estimated probability of spam falls on such extremes suggests that the model was very confident about its decisions. But what happens when the predicted and actual values differ? Using the `subset()` function, we can identify a few of these records:

```
> head(subset(sms_results, actual_type != predict_type))
  actual_type predict_type   prob_spam
53        spam          ham 0.0006796225
59        spam          ham 0.1333961018
73        spam          ham 0.3582665350
76        spam          ham 0.1224625535
81        spam          ham 0.0224863219
184       spam          ham 0.0320059616
```













Notice that the probabilities are somewhat less extreme, particularly row number 73, which the classifier felt had a 35 percent chance of being spam, yet still classified as `ham`.

The previous six examples represent six of the mistakes made by the SMS classifier. In spite of such mistakes, is the model still useful? We can answer this question by applying various error metrics to this evaluation data. In fact, many such metrics are based on a tool we've already used extensively in previous chapters.

## A closer look at confusion matrices

A **confusion matrix** is a table that categorizes predictions according to whether they match the actual value in the data. One of the table's dimensions indicates the possible categories of predicted values while the other dimension indicates the same for actual values. Although, we have only seen  $2 \times 2$  confusion matrices so far, a matrix can be created for a model predicting any number of classes. The following figure depicts the familiar confusion matrix for two-class binary model as well as the  $3 \times 3$  confusion matrix for a three-class model.

When the predicted value is the same as the actual value, this is a correct classification. Correct predictions fall on the diagonal in the confusion matrix (denoted by **O**). The off-diagonal matrix cells (denoted by **X**) indicate the cases where the predicted value differs from the actual value. These are incorrect predictions. Performance measures for classification models are based on the counts of predictions falling on and off the diagonal in these tables:

		Two Classes				Three Classes		
		Predicted Class				Predicted Class		
		A	B			A	B	C
Actual Class	A			Actual Class	A			
	B				B			
						C		

The most common performance measures consider the model's ability to discern one class versus all others. The class of interest is known as the **positive** class, while all others are known as **negative**.



The use of the terminology positive and negative is not intended to imply any value judgment (that is, good versus bad), nor does it necessarily suggest that the outcome is present or absent (that is, birth defect versus none). The choice of the positive outcome can even be arbitrary, as in cases where a model is predicting categories such as sunny versus rainy, or dog versus cat.

The relationship between positive class and negative class predictions can be depicted as a 2 x 2 confusion matrix that tabulates whether predictions fall into one of four categories:

- **True Positive (TP)**: Correctly classified as the class of interest
- **True Negative (TN)**: Correctly classified as not the class of interest
- **False Positive (FP)**: Incorrectly classified as the class of interest
- **False Negative (FN)**: Incorrectly classified as not the class of interest

For the birth defect classifier mentioned previously, the confusion matrix would tabulate whether the model's predicted birth defect status matches the patient's actual birth defect status, as shown in the following diagram:

		Predicted Birth Defect	
		no	yes
Actual Birth Defect	no	<div>TN</div> <div>True Negative</div>	<div>FP</div> <div>False Positive</div>
	yes	<div>FN</div> <div>False Negative</div>	<div>TP</div> <div>True Positive</div>

## Using confusion matrices to measure performance

With the 2 x 2 confusion matrix, we can formalize our definition of prediction **accuracy** (sometimes called the success rate) as:

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

In this formula, the terms *TP*, *TN*, *FP*, and *FN* refer to the number of times the model's predictions fell into each of these categories. Therefore, the accuracy is the proportion that represents the number of true positives and true negatives divided by the total number of predictions.

The **error rate**, or the proportion of incorrectly classified examples, is specified as:

$$\text{error rate} = \frac{FP + FN}{TP + TN + FP + FN} = 1 - \text{accuracy}$$

Notice that the error rate can be calculated as one minus the accuracy. Intuitively, this makes sense; a model that is correct 95 percent of the time is incorrect 5 percent of the time.

A quick-and-dirty way to tabulate a confusion matrix is to use the `table()` function. It's easy to remember, and will count the number of occurrences of each combination of values—exactly what we need for a confusion matrix. The command for creating a confusion matrix for the SMS data is shown as follows. The counts in this table could then be used to calculate accuracy and other statistics:

```
> table(sms_results$actual_type, sms_results$predict_type)
      ham spam
ham  1202    5
spam   29  154
```

If you would like to create a confusion matrix with more detailed output, the `CrossTable()` function in the `gmodels` package offers a highly-customizable solution. If you recall, we first used this function in *Chapter 2, Managing and Understanding Data*. However, if you didn't install the package at that time, you will need to do so using the command `install.packages("gmodels")`.

By default, the `CrossTable()` output includes proportions in each cell that indicate that cell's count as a percentage of the row, column, or total for the table. It also includes row and column totals. As shown in the following code, the syntax is similar to the `table()` function:

```
> library(gmodels)
> CrossTable(sms_results$actual_type, sms_results$predict_type)
```

The result is confusion matrix with much more details:



Cell Contents	
	N
Chi-square contribution	
N / Row Total	
N / Col Total	
N / Table Total	

Total Observations in Table: 1390

sms_results\$actual_type	sms_results\$predict_type		Row Total
	ham	spam	
ham	1202	5	1207
	16.565	128.248	
	0.996	0.004	0.868
	0.976	0.031	
	0.865	0.004	
spam	29	154	183
	109.256	845.876	
	0.158	0.842	0.132
	0.024	0.969	
	0.021	0.111	
Column Total		1231	1390
	0.886	0.114	

We've used `CrossTable()` in several previous chapters, so by now you should be familiar with the output. If you don't remember, you can refer to the table's key (labeled `Cell Contents`), which provides a description of each number in the table.

We can use the contingency table to obtain the accuracy and error rate. Since accuracy is  $(TP + TN) / (TP + TN + FP + FN)$ , we can calculate:

```
> (154 + 1202) / (154 + 1202 + 5 + 29)
[1] 0.9755396
```

We can also calculate the error rate,  $(FP + FN) / (TP + TN + FP + FN)$  as:

```
> (5 + 29) / (154 + 1202 + 5 + 29)
[1] 0.02446043
```

This is the same as one minus accuracy:

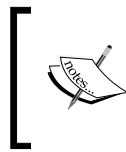
```
> 1 - 0.9755396
[1] 0.0244604
```

Although these calculations may seem simple, it can be a helpful exercise to practice thinking about how the components of the confusion matrix relate to one another. In the next section, you will see how these same pieces can be combined in different ways to create a variety of additional performance measures.

## Beyond accuracy – other measures of performance

A comprehensive description of every performance measure is not feasible. Countless measures have been developed and used for specific purposes in disciplines as diverse as medicine, information retrieval, marketing, and signal detection theory, among others. Instead, we'll consider only some of the most commonly-cited measures in machine learning literature.

The Classification and Regression Training (`caret`) package by *Max Kuhn* includes functions for computing many such performance measures. This package provides a large number of tools for preparing, training, evaluating, and visualizing machine learning models and data. In addition to its application here, we will also employ `caret` extensively in *Chapter 11, Improving Model Performance*. Before proceeding, install the package using the command `install.packages("caret")`.



For more information on `caret`, please refer to the publication: *Building predictive models in R using the caret package*, *Journal of Statistical Software*, Vol. 28, Iss. 5, by *Max Kuhn* (2008).

The `caret` package adds yet another function for creating a confusion matrix. As shown in the following commands, the syntax is similar to `table()`, but the positive outcome must be specified. Because the SMS classifier is intended to detect spam, we will set `positive = "spam"`.

```
> library(caret)
> confusionMatrix(sms_results$predict_type, sms_results$actual_type,
  positive = "spam")
```

This results in the following output:

```

Confusion Matrix and Statistics

          Reference
Prediction ham spam
ham      1202   29
spam       5  154

      Accuracy : 0.9755
      95% CI   : (0.966, 0.983)
No Information Rate : 0.8683
P-Value [Acc > NIR] : < 2.2e-16

      Kappa : 0.8867
McNemar's Test P-Value : 7.998e-05

      Sensitivity : 0.8415
      Specificity : 0.9959
Pos Pred Value : 0.9686
Neg Pred Value : 0.9764
Prevalence : 0.1317
Detection Rate : 0.1108
Detection Prevalence : 0.1144

      'Positive' Class : spam

```

The output includes a confusion matrix and a set of performance measures. Let's take a look at a few of the most commonly used statistics.


## The kappa statistic

The **kappa statistic** (labeled `Kappa` in the previous output) adjusts accuracy by accounting for the possibility of a correct prediction by chance alone. Kappa values range to a maximum value of 1, which indicates perfect agreement between the model's predictions and the true values—a rare occurrence. Values less than one indicate imperfect agreement.

Depending on how your model is to be used, the interpretation of the kappa statistic might vary. One common interpretation is shown as follows:


- Poor agreement = Less than 0.20
- Fair agreement = 0.20 to 0.40
- Moderate agreement = 0.40 to 0.60
- Good agreement = 0.60 to 0.80
- Very good agreement = 0.80 to 1.00

It's important to note, however, that these categories are subjective. While "good agreement" may be more than adequate for predicting someone's favorite ice cream flavor, "very good agreement" may not suffice if your goal is to land a shuttle safely on the surface of the moon.

 For more information on the previous scale, refer to: *The measurement of observer agreement for categorical data*, *Biometrics* Vol. 33, pp.159-174, by J.R. Landis and G.G. Koch (1977).

The following is the formula for calculating the kappa statistic. In this formula,  $Pr$  refers to the proportion of actual ( $a$ ) and expected ( $e$ ) agreement between the classifier and the true values:

$$k = \frac{\Pr(a) - \Pr(e)}{1 - \Pr(e)}$$

 There is more than one way to define the kappa statistic. The most common method, described here, uses Cohen's kappa coefficient, as described in the paper: *A coefficient of agreement for nominal scales*, *Education and Psychological Measurement* Vol. 20, pp. 37-46, by J. Cohen (1960).

These proportions are easy to obtain from a confusion matrix once you know where to look. Let's consider the confusion matrix for the SMS classification model created with the `CrossTable()` function, duplicated as follows:

sms_results\$actual_type	sms_results\$predict_type		Row Total
	ham	spam	
ham	1202	5	1207
	16.565	128.248	
	0.996	0.004	0.868
	0.976	0.031	
	0.865	0.004	
spam	29	154	183
	109.256	845.876	
	0.158	0.842	0.132
	0.024	0.969	
	0.021	0.111	
Column Total	1231	159	1390
	0.886	0.114	

Remember that the bottom value in each cell indicates the proportion of all instances falling into that cell. Therefore, to calculate the observed agreement  $Pr(a)$ , we simply add the proportion of all instances where the predicted type and actual SMS type agree. Thus, we can calculate  $Pr(a)$  as:

```
> pr_a <- 0.865 + 0.111
> pr_a
[1] 0.976
```

For this classifier, the observed and actual values agree 97.6 percent of the time – you will note that this is the same as the accuracy. The kappa statistic adjusts the accuracy relative to the expected agreement,  $Pr(e)$ , which is the probability that chance alone would lead the predicted and actual values to match, under the assumption that both are selected randomly according to the observed proportions.

To find these observed proportions, we can use the probability rules we learned in *Chapter 4, Probabilistic Learning – Classification Using Naive Bayes*. Assuming two events are independent (meaning one does not affect the other), probability rules note that the probability of both occurring is equal to the product of the probabilities of each one occurring. For instance, we know that the probability of both choosing ham is:

$$Pr(actual\_type \text{ is ham}) * Pr(predicted\_type \text{ is ham})$$

And the probability of both choosing spam is:

$$Pr(actual\_type \text{ is spam}) * Pr(predicted\_type \text{ is spam})$$

The probability that the predicted or actual type is spam or ham can be obtained from the row or column totals. For instance,  $Pr(actual\_type \text{ is ham}) = 0.868$ .

$Pr(e)$  can be calculated as the sum of the probabilities that either the predicted and actual values agree that the message is spam, or they agree that the message is ham. Since the probability of either of two mutually exclusive events (that is, they cannot happen simultaneously) occurring is equal to the sum of their probabilities, we simply add both products. In R code, this would be:

```
> pr_e <- 0.868 * 0.886 + 0.132 * 0.114
> pr_e
[1] 0.784096
```

Since  $pr\_e$  is 0.784096, by chance alone we would expect the observed and actual values to agree about 78.4 percent of the time.

This means that we now have all the information needed to complete the kappa formula. Plugging the `pr_a` and `pr_e` values into the kappa formula, we find:

```
> k <- (pr_a - pr_e) / (1 - pr_e)
> k
[1] 0.8888395
```

The kappa is about 0.89, which agrees with the previous `confusionMatrix()` output (the small difference is due to rounding). Using the suggested interpretation, we note that there is very good agreement between the classifier's predictions and the actual values.

There are a couple of R functions to calculate kappa automatically. The `Kappa()` function (be sure to note the capital  $\kappa$ ) in the Visualizing Categorical Data (`vcd`) package uses a confusion matrix of predicted and actual values. After installing the package using the command `install.packages("vcd")`, the following commands can be used to obtain kappa:

```
> library(vcd)
> Kappa(table(sms_results$actual_type, sms_results$predict_type))
               value      ASE
Unweighted 0.8867172 0.01918876
Weighted   0.8867172 0.01587936
```

We're interested in the unweighted kappa. The value 0.89 matches what we expected.



The weighted kappa is used when there are varying degrees of agreement. For example, using a scale of cold, warm, and hot, a value of warm agrees more with hot than it does with the value of cold. In the case of a two-outcome event, such as spam and ham, the weighted and unweighted kappa statistics will be identical.

The `kappa2()` function in the Inter-Rater Reliability (`irr`) package can be used to calculate kappa from vectors of predicted and actual values stored in a data frame. After installing the package using the command `install.packages("irr")`, the following commands can be used to obtain kappa:

```
> library(irr)
> kappa2(sms_results[1:2])
Cohen's Kappa for 2 Raters (Weights: unweighted)
```

```

Subjects = 1390
Raters = 2
Kappa = 0.887

z = 33.2
p-value = 0

```

In both cases, the same kappa statistic is reported, so use whichever option you are more comfortable with.



Be careful not to use the built-in `kappa()` function. It is unrelated to the Kappa statistic reported previously.

## Sensitivity and specificity

Classification often involves a balance between being overly conservative and overly aggressive in decision making. For example, an e-mail filter could guarantee to eliminate every spam message by aggressively eliminating nearly every ham message at the same time. On the other hand, a guarantee that no ham messages will be inadvertently filtered might allow an unacceptable amount of spam to pass through the filter. This tradeoff is captured by a pair of measures: sensitivity and specificity.

The **sensitivity** of a model (also called the true positive rate), measures the proportion of positive examples that were correctly classified. Therefore, as shown in the following formula, it is calculated as the number of true positives divided by the total number of positives in the data—those correctly classified (the true positives), as well as those incorrectly classified (the false negatives).

$$\text{sensitivity} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

The **specificity** of a model (also called the true negative rate), measures the proportion of negative examples that were correctly classified. As with sensitivity, this is computed as the number of true negatives divided by the total number of negatives—the true negatives plus the false positives.

$$\text{specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

Given the confusion matrix for the SMS classifier, we can easily calculate these measures by hand. Assuming that spam is the positive class, we can confirm that the numbers in the `confusionMatrix()` output are correct. For example, the calculation for sensitivity is:

```
> sens <- 154 / (154 + 29)
> sens
[1] 0.8415301
```

Similarly, for specificity we can calculate:

```
> spec <- 1202 / (1202 + 5)
> spec
[1] 0.9958575
```

The `caret` package provides functions for calculating sensitivity and specificity directly from vectors of predicted and actual values. Be careful to specify the positive or negative parameter appropriately, as shown in the following lines:

```
> library(caret)
> sensitivity(sms_results$predict_type, sms_results$actual_type,
             positive = "spam")
[1] 0.8415301
> specificity(sms_results$predict_type, sms_results$actual_type,
             negative = "ham")
[1] 0.9958575
```

Sensitivity and specificity range from 0 to 1, with values close to 1 being more desirable. Of course, it is important to find an appropriate balance between the two—a task that is often quite context-specific.

For example, in this case the sensitivity of 0.842 implies that 84 percent of spam messages were correctly classified. Similarly, the specificity of 0.996 implies that 99.6 percent of non-spam messages were correctly classified, or alternatively, 0.4 percent of valid messages were rejected as spam. The idea of rejecting 0.4 percent of valid SMS messages may be unacceptable, or it may be a reasonable tradeoff given the reduction in spam.

Use sensitivity and specificity to provide a tool for thinking about such tradeoffs. Typically, changes are made to the model, and different models are tested until finding one that meets a desired sensitivity and specificity threshold. Visualizations, such as those discussed later in this chapter, can also assist with understanding the tradeoff between sensitivity and specificity.



## Precision and recall

Closely related to sensitivity and specificity are two other performance measures, related to compromises made in classification: precision and recall. Used primarily in the context of information retrieval, these statistics are intended to provide an indication of how interesting and relevant a model's results are, or whether the predictions are diluted by meaningless noise.

The **precision** (also known as the positive predictive value) is defined as the proportion of positive examples that are truly positive; in other words, when a model predicts the positive class, how often is it correct? A precise model will only predict the positive class in cases very likely to be positive. It will be very trustworthy.

Consider what would happen if the model was very imprecise. Over time, the results would be less likely to be trusted. In the context of information retrieval, this would be similar to a search engine such as Google returning unrelated results. Eventually users would switch to a competitor such as Bing. In the case of the SMS spam filter, high precision means that the model is able to carefully target only the spam while ignoring the ham.

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

On the other hand, recall is a measure of how complete the results are. As shown in the following formula, this is defined as the number of true positives over the total number of positives. You may recognize that this is the same as sensitivity, only the interpretation differs. A model with high recall captures a large portion of the positive examples, meaning that it has wide breadth. For example, a search engine with high recall returns a large number of documents pertinent to the search query. Similarly, the SMS spam filter has high recall if the majority of spam messages are correctly identified.

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

We can calculate precision and recall from the confusion matrix. Again, assuming that spam is the positive class, the precision is:

```
> prec <- 154 / (154 + 5)
> prec
[1] 0.9685535
```

And the recall is:

```
> rec <- 154 / (154 + 29)
> rec
[1] 0.8415301
```

The `caret` package can be used to compute either of these measures from vectors of predicted and actual classes. Precision uses the `posPredValue()` function:

```
> library(caret)
> posPredValue(sms_results$predict_type, sms_results$actual_type,
               positive = "spam")
[1] 0.9685535
```

While recall uses the `sensitivity()` function as we had done before.

Similar to the inherent tradeoff between sensitivity and specificity, for most real-world problems, it is difficult to build a model with both high precision and high recall. It is easy to be precise if you target only the low-hanging fruit – the easy to classify examples. Similarly, it is easy for a model to have high recall by casting a very wide net, meaning that the model is overly aggressive at predicting the positive cases. In contrast, having both high precision and recall at the same time is very challenging. It is therefore important to test a variety of models in order to find the combination of precision and recall that meets the needs of your project.

## The F-measure

A measure of model performance that combines precision and recall into a single number is known as the **F-measure** (also sometimes called the F1 score or the F-score). The F-measure combines precision and recall using the harmonic mean. The harmonic mean is used rather than the more common arithmetic mean since both precision and recall are expressed as proportions between zero and one. The following is the formula for F-measure:

$$\text{F-measure} = \frac{2 \times \text{precision} \times \text{recall}}{\text{recall} + \text{precision}} = \frac{2 \times \text{TP}}{2 \times \text{TP} + \text{FP} + \text{FN}}$$

To calculate the F-measure, use the precision and recall values computed previously:

```
> f <- (2 * prec * rec) / (prec + rec)
> f
[1] 0.9005848
```

This is the same as using the counts from the confusion matrix:

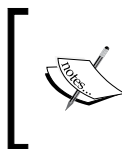
```
> f2 <- (2 * 154) / (2 * 154 + 5 + 29)
> f2
[1] 0.9005848
```

Since the F-measure reduces model performance to a single number, it provides a convenient way to compare several models side-by-side. However, this assumes that equal weight should be assigned to precision and recall, an assumption that is not always valid. It is possible to calculate F-scores using different weights for precision and recall, but choosing the weights can be tricky at best and arbitrary at worst. A better practice is to use measures such as the F-score in combination with methods that consider a model's strengths and weaknesses more globally, such as those described in the next section.

## Visualizing performance tradeoffs

Visualizations are often helpful for understanding how the performance of machine learning algorithms varies from situation to situation. Rather than thinking about a single pair of statistics such as sensitivity and specificity, or precision and recall, visualizations allow you to examine how measures vary across a wide range of values. They also provide a method for comparing learners side-by-side in a single chart.

The `ROCR` package provides an easy-to-use suite of functions for creating visualizations of the performance statistics of classification models. It includes functions for computing a large set of the most common performance measures and visualizations. The `ROCR` website, <http://rocr.bioinf.mpi-sb.mpg.de/>, includes a list of the full set of features as well as several examples of the visualization capabilities. Before continuing, install the package using the command `install.packages("ROCR")`.



For more information on the development of `ROCR`, see:  
*ROCR: visualizing classifier performance in R, Bioinformatics Vol. 21*, pp. 3940-3941, by T. Sing, O. Sander, N. Beerenwinkel, and T. Lengauer (2005).

To create visualizations with `ROCR`, two vectors of data are needed. The first must contain the class values predicted, and the second must contain the estimated probability of the positive class. These are used to create a prediction object that can be examined through plotting functions of `ROCR`.

The prediction object for the SMS classifier uses the classifier's estimated spam probabilities (`prob_spam`), and the actual class labels (`actual_type`). These are combined using the `prediction()` function in the following lines:

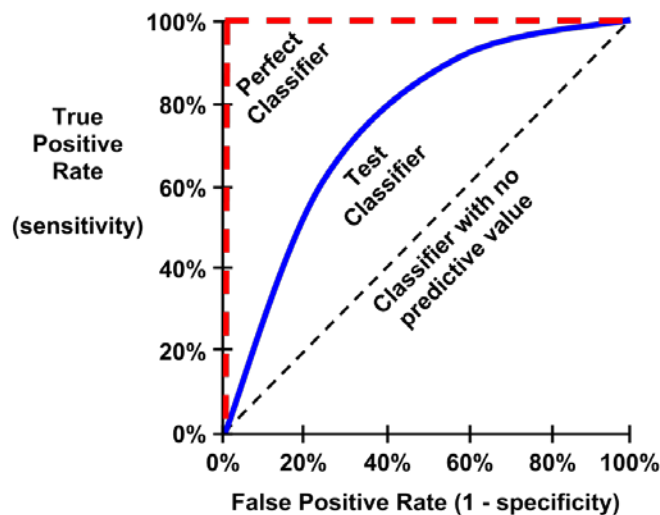
```
> library(ROCR)
> pred <- prediction(predictions = sms_results$prob_spam,
                     labels = sms_results$actual_type)
```

ROCR provides a `performance()` function for computing measures of performance on prediction objects such as `pred`, which was used in previous code example. The resulting performance object can be visualized using the `R.plot()` function. Given these three functions, a large variety of depictions can be created.

## ROC curves

The **ROC curve (Receiver Operating Characteristic)** is commonly used to examine the tradeoff between the detection of true positives, while avoiding the false positives. As you might suspect from the name, ROC curves were developed by engineers in the field of communications around the time of World War II; receivers of radar and radio signals needed a method to discriminate between true signals and false alarms. The same technique is useful today for visualizing the efficacy of machine learning models.

The characteristics of a typical ROC diagram are depicted in the following plot. Curves are defined on a plot with the proportion of true positives on the vertical axis, and the proportion of false positives on the horizontal axis. Because these values are equivalent to **sensitivity** and **(1 - specificity)**, respectively, the diagram is also known as a sensitivity/specificity plot:




The points comprising ROC curves indicate the true positive rate at varying false positive thresholds. To create the curves, a classifier's predictions are sorted by the model's estimated probability of the positive class, with the largest values first. Beginning at the origin, each prediction's impact on the true positive rate and false positive rate will result in a curve tracing vertically (for a correct prediction), or horizontally (for an incorrect prediction).

To illustrate this concept, three hypothetical classifiers are contrasted in the previous plot. First, the diagonal line from the bottom-left to the top-right corner of the diagram represents a *classifier with no predictive value*. This type of classifier detects true positives and false positives at exactly the same rate, implying that the classifier cannot discriminate between the two. This is the baseline by which other classifiers may be judged; ROC curves falling close to this line indicate models that are not very useful. Similarly, the *perfect classifier* has a curve that passes through the point at 100 percent true positive rate and 0 percent false positive rate. It is able to correctly identify all of the true positives before it incorrectly classifies any negative result. Most real-world classifiers are similar to the *test classifier*; they fall somewhere in the zone between perfect and useless.

The closer the curve is to the perfect classifier, the better it is at identifying positive values. This can be measured using a statistic known as the **area under the ROC curve** (abbreviated AUC). The AUC, as you might expect, treats the ROC diagram as a two-dimensional square and measures the total area under the ROC curve. AUC ranges from 0.5 (for a classifier with no predictive value), to 1.0 (for a perfect classifier). A convention for interpreting AUC scores uses a system similar to academic letter grades:

- $0.9 - 1.0 = A$  (outstanding)
- $0.8 - 0.9 = B$  (excellent/good)
- $0.7 - 0.8 = C$  (acceptable/fair)
- $0.6 - 0.7 = D$  (poor)
- $0.5 - 0.6 = F$  (no discrimination)

As with most scales similar to this, the levels may work better for some tasks than others; the categorization is somewhat subjective.

 It's also worth noting that two ROC curves may be shaped very differently, yet have identical AUC. For this reason, AUC can be extremely misleading. The best practice is to use AUC in combination with qualitative examination of the ROC curve.

Creating ROC curves with the `ROCR` package involves building a performance object for the `pred` prediction object we computed earlier. Since ROC curves plot true positive rates versus false positive rates, we simply call the `performance()` function while specifying the `tpr` and `fpr` measures, as shown in the following code:

```
> perf <- performance(pred, measure = "tpr", x.measure = "fpr")
```

Using the `perf` performance object, we can visualize the ROC curve with R's `plot()` function. As shown in the following code lines, many of the standard parameters for adjusting the visualization can be used, such as `main` (for adding a title), `col` (for changing the line color), and `lwd` (for adjusting the line width):

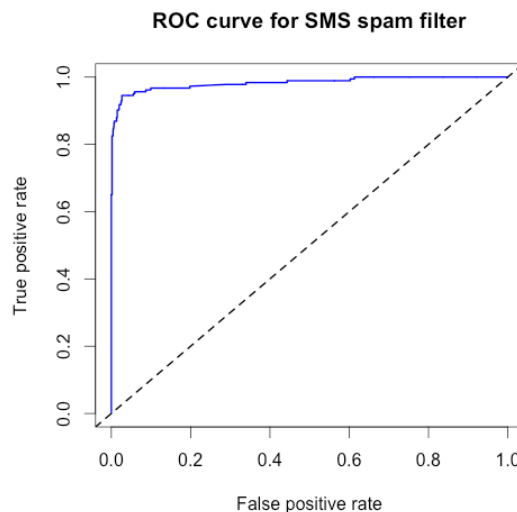
```
> plot(perf, main = "ROC curve for SMS spam filter",  
      col = "blue", lwd = 3)
```

Although the `plot()` command, used in previous lines of code, is sufficient to create a valid ROC curve, it is helpful to add a reference line to indicate the performance of a classifier with no predictive value.

For plotting such a line, we'll use the `abline()` function. This function can be used to specify a line in slope-intercept form, where `a` is the intercept and `b` is the slope. Since we need an identity line that passes through the origin, we'll set the intercept to `a=0` and the slope to `b=1` as shown in the following plot. The `lwd` parameter adjusts the line thickness, while the `lty` parameter adjusts the type of line. For example, `lty = 2` indicates a dashed line.

```
> abline(a = 0, b = 1, lwd = 2, lty = 2)
```

The end result is an ROC plot with a dashed reference line:



Qualitatively, we can see that this ROC curve appears to occupy the space in the top-left corner of the diagram, which suggests that it is closer to a perfect classifier than the dashed line indicating a useless classifier. To confirm this quantitatively, we can use the `ROCR` package to calculate the AUC. To do so, we first need to create another performance object, this time specifying `measure = "auc"`, as shown in the following code:

```
> perf.auc <- performance(pred, measure = "auc")
```

Since `perf.auc` is an object (specifically known as an S4 object) we need to use a special type of notation to access the values stored within. S4 objects hold information in positions known as slots. The `str()` function can be used to see all slots for an object:

```
> str(perf.auc)
Formal class 'performance' [package "ROCR"] with 6 slots
 ..@ x.name      : chr "None"
 ..@ y.name      : chr "Area under the ROC curve"
 ..@ alpha.name  : chr "none"
 ..@ x.values    : list()
 ..@ y.values    :List of 1
 .. ..$ : num 0.983
 ..@ alpha.values: list()
```

Notice that slots are prefixed with the `@` symbol. To access the AUC value, which is stored as a list in the `y.values` slot, we can use the `@` notation along with the `unlist()` function, which simplifies lists to a vector of numeric values:

```
> unlist(perf.auc@y.values)
[1] 0.9829999
```

The AUC for the SMS classifier is 0.98, which is extremely high. But how do we know whether the model is just as likely to perform well on another dataset? In order to answer such questions, we need to better understand how far we can extrapolate a model's predictions beyond the test data.

## Estimating future performance

Some R machine learning packages present confusion matrices and performance measures during the model building process. The purpose of these statistics is to provide insight about the model's **resubstitution error**, which occurs when the training data is incorrectly predicted in spite of the model being built directly from this data. This information is intended to be used as a rough diagnostic, particularly to identify obviously poor performers.

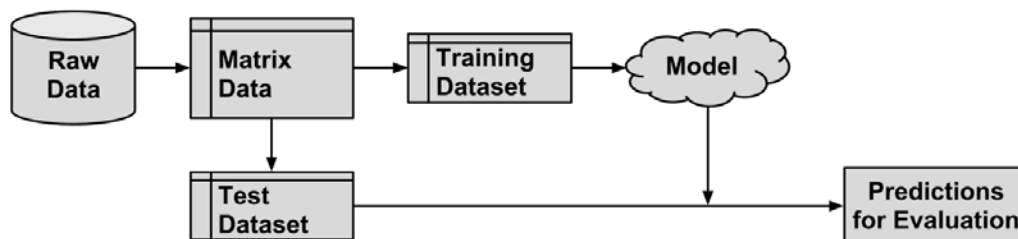
The resubstitution error is not a very useful marker of future performance, however. For example, a model that used rote memorization to perfectly classify every training instance (that is, zero resubstitution error) would be unable to make predictions on data it has never seen before. For this reason, the error rate on the training data can be extremely optimistic about a model's future performance.

Instead of relying on resubstitution error, a better practice is to evaluate a model's performance on data it has not yet seen. We used such a method in previous chapters when we split the available data into a set for training and a set for testing. In some cases, however, it is not always ideal to create training and test datasets. For instance, in a situation where you have only a small pool of data, you might not want to reduce the sample any further by dividing it into training and test sets.

Fortunately, there are other ways to estimate a model's performance on unseen data. The `caret` package that we used previously for calculating performance measures also, offers a number of functions for this purpose. If you are following along with the R code examples and haven't already installed the `caret` package, please do so. You will also need to load the package to the R session using the `library(caret)` command.

## The holdout method

The procedure of partitioning data into training and test datasets that we used in previous chapters is known as the **holdout method**. As shown in the following diagram, the **training dataset** is used to generate the model, which is then applied to the **test dataset** to generate predictions for evaluation. Typically, about one-third of the data is held out for testing and two-thirds used for training, but this proportion can vary depending on the amount of data available. To ensure that the training and test data do not have systematic differences, examples are randomly divided into the two groups.





For the holdout method to result in a truly accurate estimate of future performance, at no time should results from the test dataset be allowed to influence the model. It is easy to unknowingly violate this rule by choosing a best model based upon the results of repeated testing. Instead, it is better to divide the original data so that in addition to the training and test datasets, a third validation dataset is available. The validation dataset would be used for iterating and refining the model or models chosen, leaving the test dataset to be used only once as a final step to report an estimated error rate for future predictions. A typical split between training, test, and validation would be 50 percent, 25 percent, and 25 percent respectively.



A keen reader will note that holdout test data was used in previous chapters to compare several models. This would indeed violate the rule as stated previously, and therefore the test data might have been more accurately termed validation data. If we use test data to make a decision, we are cherry-picking results and the evaluation is no longer an unbiased estimate of future performance.

A simple method for creating holdout samples uses random number generators to assign records to partitions. This technique was first used in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules* to create training and test datasets.



If you'd like to follow along with the following examples, download the dataset `credit.csv` from the Packt Publishing's website and load to a data frame using the command `credit <- read.csv("credit.csv")`.

Suppose we have a data frame named `credit` with 1000 rows of data. We can divide this into three partitions:

```
> random_ids <- order(runif(1000))
> credit_train <- credit[random_ids[1:500],]
> credit_validate <- credit[random_ids[501:750], ]
> credit_test <- credit[random_ids[751:1000], ]
```

The first line creates a vector of randomly ordered row IDs from 1 to 1000. These IDs are then used to divide the `credit` data frame into 500, 250, and 250 records comprising the training, validation, and test datasets.


One problem with holdout sampling is that each partition may have a larger or smaller proportion of some classes. In certain cases, particularly those in which a class is a very small proportion of the dataset, this can lead a class to be omitted from the training dataset—a significant problem, because the model cannot then learn this class.

In order to reduce the chance that this will occur, a technique called **stratified random sampling** can be used. Although, on average, a random sample will contain roughly the same proportion of class values as the full dataset, stratified random sampling ensures that the generated random partitions have approximately the same proportion of each class as the full dataset.

The `caret` package provides a `createDataPartition()` function that will create partitions based on stratified holdout sampling. Code for creating a stratified sample of training and test data for the `credit` dataset is shown in the following commands. To use the function, a vector of class values must be specified (here, `default` refers to whether a loan went into default), in addition to a parameter `p`, which specifies the proportion of instances to be included in the partition. The `list = FALSE` parameter prevents the result from being stored in list format:

```
> in_train <- createDataPartition(credit$default, p = 0.75,  
  list = FALSE)  
> credit_train <- credit[in_train, ]  
> credit_test <- credit[-in_train, ]
```

The `in_train` vector indicates row numbers included in the training sample. We can use these row numbers to select examples for the `credit_train` data frame. Similarly, by using a negative symbol, we can use the rows not found in the `in_train` vector for the `credit_test` dataset.

 Since models trained on larger datasets generally perform better, a common practice is to retrain the model on the full set of data (that is, training plus test and validation) after a final model has been selected and evaluated, allowing the model maximum use of available data.

Although it distributes the classes evenly, stratified sampling does not guarantee other types of representativeness. Some samples may have too many or too few difficult cases, easy-to-predict cases, or outliers. This is especially true for smaller datasets, which may not have a large enough portion of such cases to divide among training and test sets.


In addition to potentially biased samples, another problem with the holdout method is that substantial portions of data must be reserved for testing and validating the model. Since these data cannot be used to train the model until its performance has been measured, the performance estimates are likely to be overly conservative.

A technique called **repeated holdout** is sometimes used to mitigate the problems of randomly composed training datasets. The repeated holdout method is a special case of the holdout method that uses the average result from several random holdout samples to evaluate a model's performance. As multiple holdout samples are used, it is less likely that the model is trained or tested on non-representative data. We'll expand on this idea in the next section.

## Cross-validation

The repeated holdout is the basis of a technique known as **k-fold cross-validation** (or k-fold CV), which has become the industry standard for estimating model performance. But rather than taking repeated random samples that could potentially use the same record more than once, k-fold CV randomly divides the data into  $k$  completely separate random partitions called folds.

Although  $k$  can be set to any number, by far the most common convention is to use **10-fold cross-validation** (10-fold CV). Why 10 folds? Empirical evidence suggests that there is little added benefit to using a greater number. For each of the 10 folds (each comprising 10 percent of the total data), a machine learning model is built on the remaining 90 percent of data. The fold's matching 10 percent sample is then used for model evaluation. After the process of training and evaluating the model has occurred for 10 times (with 10 different training/testing combinations), the average performance across all folds is reported.

 An extreme case of k-fold CV is the **leave-one-out method**, which performs k-fold CV using a fold for each one of the data's examples. This ensures that the greatest amount of data is used for training the model. Although this may seem useful, it is so computationally expensive that it is rarely used in practice.

Datasets for cross-validation can be created using the `createFolds()` function in the `caret` package. Similar to the stratified random holdout sampling, this function will attempt to maintain the same class balance in each of the folds as in the original dataset. The following is the command for creating 10 folds:

```
> folds <- createFolds(credit$default, k = 10)
```

The result of the `createFolds()` function is a list of vectors storing the row numbers for each of the `k = 10` requested folds. We can peek at the contents using `str()`:

```
> str(folds)
List of 10
 $ Fold01: int  [1:100] 1 5 12 13 19 21 25 32 36 38 ...
 $ Fold02: int  [1:100] 16 49 78 81 84 93 105 108 128 134 ...
 $ Fold03: int  [1:100] 15 48 60 67 76 91 102 109 117 123 ...
 $ Fold04: int  [1:100] 24 28 59 64 75 85 95 97 99 104 ...
 $ Fold05: int  [1:100] 9 10 23 27 29 34 37 39 53 61 ...
 $ Fold06: int  [1:100] 4 8 41 55 58 103 118 121 144 146 ...
 $ Fold07: int  [1:100] 2 3 7 11 14 33 40 45 51 57 ...
 $ Fold08: int  [1:100] 17 30 35 52 70 107 113 129 133 137 ...
 $ Fold09: int  [1:100] 6 20 26 31 42 44 46 63 79 101 ...
 $ Fold10: int  [1:100] 18 22 43 50 68 77 80 88 106 111 ...
```

Here, we see that the first fold is named `Fold01`, and stores 100 integers indicating the 100 rows in the `credit` data frame for the first fold. To create training and test datasets to build and evaluate a model, one more step is needed. The following commands show how to create data for the first fold. Just as we had done with stratified holdout sampling, we'll assign the selected examples to the training dataset and use the negative symbol to assign everything else to the test dataset:

```
> credit01_train <- credit[folds$Fold01, ]
> credit01_test  <- credit[-folds$Fold01, ]
```

To perform the full 10-fold CV, this step would need to be repeated a total of 10 times, building a model, and then calculating the model's performance each time. At the end, the performance measures would be averaged to obtain the overall performance. Thankfully, we can automate this task by applying several of the techniques we've learned before.

To demonstrate the process, we'll estimate the kappa statistic for a C5.0 decision tree model of the `credit` data using 10-fold CV. First, we need to load `caret` (for creating the folds), `C50` (for the decision tree), and `irr` (for calculating kappa). The latter two packages were chosen for illustrative purposes; if you desire, you can use a different model or a different performance measure with the same series of steps.

```
> library(caret)
> library(C50)
> library(irr)
```

Next, we'll create a list of 10 `folds` as we have done previously. The `set.seed()` function is used here to ensure that the results are consistent if you run the same code again:

```
> set.seed(123)
> folds <- createFolds(credit$default, k = 10)
```

Finally, we will apply a series of identical steps to the list of folds using the `lapply()` function. As shown in the following code, because there is no existing function that does exactly what we need, we must define our own function to pass to `lapply()`. Our custom function divides the `credit` data frame into training and test data, creates a decision tree using the `C5.0()` function on the training data, generates a set of predictions from the test data, and compares the predicted and actual values using the `kappa2()` function:

```
> cv_results <- lapply(folds, function(x) {
  credit_train <- credit[x, ]
  credit_test <- credit[-x, ]
  credit_model <- C5.0(default ~ ., data = credit_train)
  credit_pred <- predict(credit_model, credit_test)
  credit_actual <- credit_test$default
  kappa <- kappa2(data.frame(credit_actual, credit_pred))$value
  return(kappa)
})
```

The resulting kappa statistics are compiled into a list stored in the `cv_results` object, which we can examine using `str()`:

```
> str(cv_results)
List of 10
 $ Fold01: num 0.283
 $ Fold02: num 0.108
 $ Fold03: num 0.326
 $ Fold04: num 0.162
 $ Fold05: num 0.243
 $ Fold06: num 0.257
 $ Fold07: num 0.0355
 $ Fold08: num 0.0761
 $ Fold09: num 0.241
 $ Fold10: num 0.253
```

In this way, we've transformed our list of IDs for 10 folds into a list of kappa statistics. There's just one more step remaining: we need to calculate the average of these 10 values. Although you will be tempted to type `mean(cv_results)`, because `cv_results` is not a numeric vector the result would be an error. Instead, use the `unlist()` function, which eliminates the list structure and reduces `cv_results` to a numeric vector. From there, we can calculate the mean kappa as expected:

```
> mean(unlist(cv_results))  
[1] 0.1984929
```

Unfortunately, this kappa statistic is fairly low – in fact, this corresponds to "poor" on the interpretation scale – which suggests that the credit scoring model does not perform much better than random chance. In the next chapter, we'll examine automated methods based on 10-fold CV that can assist us with improving the performance of this model.



Perhaps the current gold standard method for reliably estimating model performance is repeated k-fold CV. As you might guess from the name, this involves repeatedly applying k-fold CV and averaging the results. A common strategy is to perform 10-fold CV ten times. Although computationally intensive, this provides a very robust estimate.

## Bootstrap sampling

A slightly less popular, but still fairly widely-used alternative to k-fold CV is known as **bootstrap sampling**, the **bootstrap**, or **bootstrapping** for short. Generally speaking, these refer to statistical methods of using random samples of data to estimate properties of a larger set. When this principle is applied to machine learning model performance, it implies the creation of several randomly-selected training and test datasets, which are then used to estimate performance statistics. The results from the various random datasets are then averaged to obtain a final estimate of future performance.

So, what makes this procedure different from k-fold CV? Where cross-validation divides the data into separate partitions, in which each example can appear only once, the bootstrap allows examples to be selected multiple times through a process of sampling with replacement. This means that from the original dataset of  $n$  examples, the bootstrap procedure will create one or more new training datasets that also contain  $n$  examples, some of which are repeated. The corresponding test datasets are then constructed from the set of examples that were not selected for the respective training datasets.

Using sampling with replacement as described previously, the probability that any given instance is included in the training dataset is 63.2 percent. Consequently, the probability of any instance being in the test dataset is 36.8 percent. In other words, the training data represents only 63.2 percent of available examples, some of which are repeated. In contrast with 10-fold CV, which uses 90 percent of examples for training, the bootstrap sample is less representative of the full dataset.

As a model trained on only 63.2 percent of the training data is likely to perform worse than a model trained on a larger training set, the bootstrap's performance estimates may be substantially lower than what will be obtained when the model is later trained on the full dataset. A special case of bootstrapping known as the **0.632 bootstrap** accounts for this by calculating the final performance measure as a function of performance on both the training data (which is overly optimistic) and the test data (which is overly pessimistic). The final error rate is then estimated as:

$$\text{error} = 0.632 \times \text{error}_{\text{test}} + 0.368 \times \text{error}_{\text{train}}$$

One advantage of the bootstrap over cross-validation is that it tends to work better with very small datasets. Additionally, bootstrap sampling has applications beyond performance measurement. In particular, in the following chapter we'll learn how the principles of bootstrap sampling can be used to improve model performance.

## Summary

This chapter presented a number of the most common measures and techniques for evaluating the performance of machine learning classification models. Although accuracy provides a simple method for examining how often a model is correct, this can be misleading in the case of rare events because the real-life cost of such events may be inversely proportional to how frequently they appear in the data.

A number of measures based on confusion matrices better capture the balance among the costs of various types of errors. Closely examining the tradeoffs between sensitivity and specificity or precision and recall can be a useful tool for thinking about the implications of errors in the real world. Visualizations such as the ROC curve are also helpful toward this end.

It is also worth mentioning that sometimes the best measure of a model's performance is to consider how well it meets or doesn't meet other objectives. For instance, you may need to explain a model's logic in simple language, which would eliminate some models from consideration. Additionally, even if it performs very well, a model that is too slow or difficult to scale to a production environment is completely useless.

An obvious extension of measuring performance is to identify automated ways to find the best models for a particular task. In the next chapter, we will build upon our work so far to investigate ways to make smarter models by systematically iterating, refining, and combining learning algorithms.