CHAPTER 4

# Classification

We are entering the realm of predictive analytics—the process in which historical records are used to make a prediction about an uncertain future. At a very fundamental level, most predictive analytics problems can be categorized into either classification or numeric prediction problems. In classification or class prediction, we try to use the information from the predictors or independent variables to sort the data samples into two or more distinct *classes* or *buckets*. In the case of numeric prediction, we try to predict the numeric value of a dependent variable using the values assumed by the independent variables, as is done in a traditional regression modeling.

Let us describe the classification process with a simple, fun example. Most golfers enjoy playing if the weather and outlook conditions meet certain requirements: too hot or too humid conditions, even if the outlook is sunny, are not preferred. On the other hand, overcast skies are no problem for playing even if the temperatures are somewhat cool. Based on the historic records of these conditions and preferences, and information about a day's temperature, humidity level, and outlook, classification will allow us to predict if someone prefers to play golf or not. The outcome of classification is to categorize the weather conditions when golf is likely to be played or not, quite simply: Play or Not Play (two classes). The predictors can be continuous (temperature, humidity) or categorical (sunny, cloudy, windy, etc.). Those beginning to explore predictive analytics tools are confused by the dozens of techniques that are available to address these types of classification problems. In this chapter we will explore several commonly used data mining techniques where the idea is to develop rules, relationships, and models based on predictor information that can be applied to classify outcomes from new and unseen data.

We start out with fairly simple schemes and progress to more sophisticated techniques. Each section contains essential algorithmic details about the technique, describes how it is developed using simple examples, and finally closes out with implementation details using RapidMiner.



**Predictive Analytics and Data Mining.** http://dx.doi.org/10.1016/B978-0-12-801460-8.00004-5

## 4.1 DECISION TREES

Decision trees (also known as classification trees) are probably one of the most intuitive and frequently used data mining techniques. From an analyst's point of view, they are easy to set up and from a business user's point of view they are easy to interpret. Classification trees, as the name implies, are used to separate a data set into classes belonging to the response variable. Usually the response variable has two classes: *Yes or No (1 or 0)*. If the response variable has *more* than two categories, then variants of the decision tree algorithm have been developed that may be applied (Quinlan, 1986). In either case, classification trees are used when the response or target variable is categorical in nature.

*Regression trees* (Brieman, 1984) are similar in function to classification trees and may be used for numeric prediction problems, when the response variable is numeric or continuous: for example, predicting the price of a consumer good based on several input factors. Thus regression trees are applicable for *prediction* type of problems as opposed to *classification*. Keep in mind that in either case, the predictors or independent variables may be either categorical or numeric. It is the *target variable* that determines the type of decision tree needed. (Collectively, the algorithm for classification and regression trees is referred to as CART.)

### 4.1.1 How it Works

A decision tree model takes a form of decision flowchart (or an inverted tree) where an attribute is tested in each node. At end of the decision tree path is a leaf node where a prediction is made about the target variable based on conditions set forth by the decision path. The nodes split the data set into subsets. In a decision tree, the idea is to *split* the data set based on homogeneity of data. Let us say for example we have two variables, age and weight, that predict if a person is likely to sign up for a gym membership or not. In our training data if it was seen that 90% of the people who are older than 40 signed up, we may split the data into two parts: one part consisting of people older than 40 and the other part consisting of people under 40. The first part is now "90% pure" from the standpoint of which class they belong to. However we need a rigorous measure of impurity, which meets certain criteria, based on computing a proportion of the data that belong to a class. These criteria are simple:

1. The measure of impurity of a data set must be at a maximum when all possible classes are equally represented. In our gym membership example, in the initial data set if 50% of samples belonged to "not signed up" and 50% of samples belonged to "signed up," then this nonpartitioned raw data would have maximum impurity.
2. The measure of impurity of a data set must be zero when only one class is represented. For example, if we form a group of only those people who signed up for the membership (only one class = members), then this subset has "100% purity" or "0% impurity."

Measures such as *entropy* or *Gini index* easily meet these criteria and are used to build decision trees as described in the following sections. Different criteria will build different trees through different biases, for example, *information gain* favors tree splits that contain many cases, while *information gain ratio* attempts to balance this.

## HOW PREDICTIVE ANALYTICS CAN REDUCE UNCERTAINTY IN A BUSINESS CONTEXT: THE CONCEPT OF ENTROPY

Imagine a box that can contain one of three colored balls inside—red, yellow, and blue, see Figure 4.1. Without opening the box, if you had to "predict" which colored ball is inside, you are basically dealing with lack of information or uncertainty. Now what is the *highest* number of "yes/no" questions that can be asked to reduce this uncertainty and thus increase our information?

1. Is it red? No.
2. Is it yellow? No.

Then it must be blue.

That is *two* questions. If there were a fourth color, green, then the highest number of yes/no questions is *three*. By extending this reasoning, it can be mathematically shown that the maximum number of *binary* questions needed to reduce uncertainty is essentially log(T), where the log is taken to base 2 and T is the number of possible outcomes (Meagher, 2005) (e.g., if you have only one color, i.e., one outcome, then log(1) = 0, which means there is no uncertainty!)

Many real world business problems can be thought of as extensions to this "uncertainty reduction" example. For example, knowing only a handful of characteristics such as the length of a loan, borrower's occupation, annual income, and previous credit behavior, we can use several of the available predictive analytics techniques to rank the riskiness of a potential loan, and by extension, the interest rate of the loan. This is nothing but a more sophisticated uncertainty reduction exercise, similar in spirit to the ball-in-a-box problem. Decision trees embody this problem-solving technique by systematically examining the available attributes and their impact on the eventual class or category of a sample. We will examine in detail later in this section how to predict the credit ratings of a bank's customers using their demographic and other behavioral data and using the decision tree which is a practical implementation of the entropy principle for decision making under uncertainty.


**FIGURE 4.1**
Playing 20 questions with entropy.

Continuing with the example in the box, if there are T events with equal probability of occurrence P, then T = 1/P. Claude Shannon, who developed the mathematical underpinnings for information theory (Shannon, 1948), defined entropy as log(1/P) or –log P where P is the probability of an event occurring. If the probability for all events is not identical, we need a weighted expression and thus entropy, H, is adjusted as follows:

$$H = -\sum p_k \log_2 (p_k) \tag{4.1}$$

where k = 1, 2, 3, …, m represent the m classes of the target variable. The $p_k$ represent the proportion of samples that belong to class k. For our gym membership example from earlier, there are two classes: member or nonmember. If our data set had 100 samples with 50% of each, then the entropy of the dataset is given by $H = -[(0.5 \log_2 0.5) + (0.5 \log_2 0.5)] = -\log_2 0.5 = -(-1) = 1$. On the other hand, if we can partition the data into two sets of 50 samples each that contain all members and all nonmembers, the entropy of either of these two partitioned sets is given by $H = -1 \log_2 1 = 0$. Any other proportion of samples within a data set will yield entropy values between 0 and 1.0 (which is the maximum). The Gini index (G) is similar to the entropy measure in its characteristics and is defined as

$$G = \sum (1 - p_k^2) \tag{4.2}$$

The value of G ranges between 0 and a maximum value of 0.5, but otherwise has properties identical to H, and either of these formulations can be used to create partitions in the data (Cover, 1991).

Let us go back to the example of the golf data set introduced earlier, to fully understand the application of entropy concepts for creating a decision tree. This was the same dataset used by J. Ross Quinlan to introduce one of the original decision tree algorithms, the *Iterative Dichotomizer* 3, or ID3 (Quinlan, 1986). The full data is shown in Table 4.1.

There are essentially two questions we need to answer at each step of the tree building process: *where to split the data* and *when to stop splitting*.

### *Classic Golf Example and How It Is Used to Build a Decision Tree*

Where to split data?

There are 14 examples, with four attributes—Temperature, Humidity, Wind, and Outlook. The target attribute that needs to be predicted is Play with two classes: Yes and No. We want to understand how to build a decision tree using this simple data set.

**Table 4.1** The Classic Golf Data Set

| Outlook | Temperature | Humidity | Windy | Play |
|---|---|---|---|---|
| sunny | 85 | 85 | FALSE | no |
| sunny | 80 | 90 | TRUE | no |
| overcast | 83 | 78 | FALSE | yes |
| rain | 70 | 96 | FALSE | yes |
| rain | 68 | 80 | FALSE | yes |
| rain | 65 | 70 | TRUE | no |
| overcast | 64 | 65 | TRUE | yes |
| sunny | 72 | 95 | FALSE | no |
| sunny | 69 | 70 | FALSE | yes |
| rain | 75 | 80 | FALSE | yes |
| sunny | 75 | 70 | TRUE | yes |
| overcast | 72 | 90 | TRUE | yes |
| overcast | 81 | 75 | FALSE | yes |
| rain | 71 | 80 | TRUE | no |

Start by partitioning the data on each of the four regular attributes. Let us start with Outlook. There are three categories for this variable: sunny, overcast, and rain. We see that when it is overcast, there are four examples where the outcome was Play = yes for all four cases (see Figure 4.2) and so the proportion of examples in this case is 100% or 1.0. Thus if we split the data set here, the resulting four sample partition will be 100% pure for Play = yes. Mathematically for this partition, the entropy can be calculated using Eq. 4.1 as

$$H_{\text{outlook:overcast}} = -(0/4)\log_2(0/4) - (4/4)\log_2(4/4) = 0.0$$

Similarly, we can calculate the entropy in the other two situations for Outlook:

$$H_{\text{outlook:sunny}} = -(2/5)\log_2(2/5) - (3/5)\log_2(3/5) = 0.971$$

$$H_{\text{outlook:rain}} = -(3/5)\log_2(3/5) - (2/5)\log_2(2/5) = 0.971$$

For the attribute on the whole, the total "information" is calculated as the weighted sum of these component entropies. There are four instances of Outlook = overcast, thus the proportion for overcast is given by $p_{\text{outlook:overcast}} = 4/14$. The other proportions (for Outlook = sunny and rain) are 5/14 each:

$$I_{\text{outlook}} = p_{\text{outlook:overcast}} * H_{\text{outlook:overcast}} + p_{\text{outlook:sunny}} * H_{\text{outlook:sunny}} + p_{\text{outlook:rain}} * H_{\text{outlook:rain}}$$

| Row No. | Play | Outlook |
|---|---|---|
| 3 | yes | overcast |
| 7 | yes | overcast |
| 12 | yes | overcast |
| 13 | yes | overcast |
| 4 | yes | rain |
| 5 | yes | rain |
| 6 | no | rain |
| 10 | yes | rain |
| 14 | no | rain |
| 1 | no | sunny |
| 2 | no | sunny |
| 8 | no | sunny |
| 9 | yes | sunny |
| 11 | yes | sunny |

**FIGURE 4.2**
Splitting the data on the Outlook attribute.

$$I_{outlook} = (4/14) * 0 + (5/14) * 0.971 + (5/14) * 0.971 = 0.693$$

Had we *not* partitioned the data along the three values for Outlook, the total information would have been simply the weighted average of the respective entropies for the two classes whose overall proportions were 5/14 (Play = no) and 9/14 (Play = yes):

$$I_{outlook,no\ partition} = -(5/14)\log_2(5/14) - (9/14)\log_2(9/14) = 0.940$$

By creating these splits or partitions, we have reduced some entropy (and thus gained some information). This is called, aptly enough, *information gain*. In the case of Outlook, this is given simply by

$$I_{outlook,\ no\ partition} - I_{outlook} = 0.940 - 0.693 = 0.247$$

We can now compute similar information gain values for the other three attributes, as shown in Table 4.2.

**Table 4.2** Computing the Information Gain for All Attributes

| Attribute | Information Gain |
|---|---|
| Temperature | 0.029 |
| Humidity | 0.102 |
| Wind | 0.048 |
| Outlook | 0.247 |

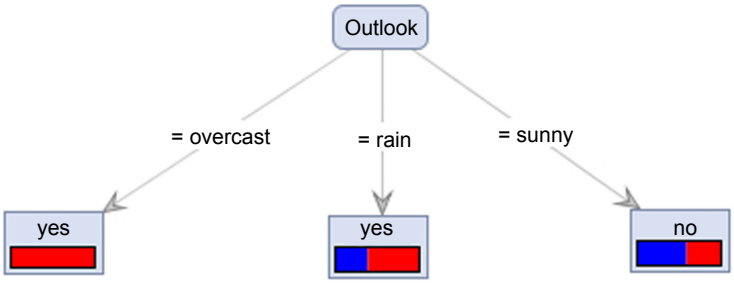



**FIGURE 4.3**
Splitting the golf data on the Outlook attribute yields three subsets or branches. The middle and right branches may be split further.

For numeric variables, possible split points to examine are essentially averages of available values. For example, the first potential split point for Humidity could be Average [65,70], which is 67.5, the next potential split point could be Average [70,75], which is 72.5, and so on. We use similar logic for the other numeric attribute, Temperature. The algorithm computes the information gain at each of these potential split points and chooses the one which maximizes it. Another way to approach this would be to discretize the numerical ranges, for example, Temperature >=80 could be considered "Hot," between 70 to 79 "Mild," and less than 70 "Cool."

From Table 4.2, it is clear that if we partition the data set into three sets along the three values of Outlook, we will experience the largest information gain. This gives the first node of the decision tree as shown in Figure 4.3. As noted earlier, the terminal node for the Outlook = overcast branch consists of four samples, all of which belong to the class Play = yes. The other two branches contain a mix of classes. The Outlook = rain branch has three yes results and the Outlook = sunny branch has three no results.

Thus not all the final partitions are 100% homogenous. This means that we could apply the same process for each of these subsets till we get "purer" results. So we revert back to the first question once again—where to split the data? Fortunately this was already answered for us when we computed the

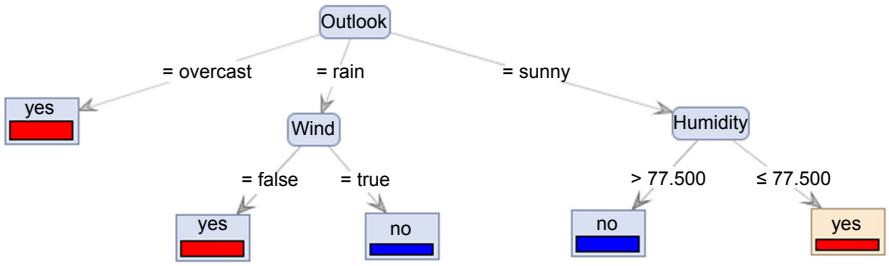



**FIGURE 4.4**
A final decision tree for the golf data.

information gain for all attributes. We simply use the other attributes that yielded the highest gains. Following the logic, we can next split the Outlook = sunny branch along Humidity (which yielded the second highest information gain) and split the Outlook = rain branch along Wind (which yielded the third highest gain). The fully grown tree shown in Figure 4.4 does precisely that.

### *Pruning a Decision Tree: When to Stop Splitting Data?*

In real world data sets, it is very unlikely that we will get terminal nodes that are 100% homogeneous as we just saw for the golf data set. In this case, we will need to instruct the algorithm when to stop. There are several situations where we can terminate the process:

- No attribute satisfies a minimum information gain threshold (such as the one computed in Table 4.2).
- A maximal depth is reached: as the tree grows larger, not only does interpretation get harder, but we run into a situation called "overfitting."
- There are less than a certain number of examples in the current subtree: again a mechanism to prevent *overfitting*.

So what exactly is overfitting? Overfitting occurs when a model tries to memorize the training data instead of generalizing the relationship between inputs and output variables. Overfitting often has the effect of performing very well on the training data set, but performing poorly on any new data previously unseen by the model. As mentioned above, overfitting by a decision tree results not only in a difficult to interpret model, but also provides quite a useless model for unseen data. To prevent overfitting, we may need to restrict tree growth or reduce it, using a process called *pruning*. All of the three stopping techniques mentioned above constitute what is known as *pre-pruning* the decision tree, because the pruning occurs before or during the growth of the tree. There are also methods that will not restrict the number of branches and allow

the tree to grow as deep as the data will allow, and *then* trim or prune those branches that do not effectively change the classification error rates. This is called *post-pruning*. Post-pruning may sometimes be a better option because we will not miss any small but potentially significant relationships between attribute values and classes if we allow the tree to reach its maximum depth. However, one drawback with post-pruning is that it requires additional computations, which may be wasted when the tree needs to be trimmed back.

We can now summarize the application of the decision tree algorithm as the following simple five-step process:

1. Using Shannon entropy, sort the data set into homogenous (by class) and nonhomogenous variables. Homogenous variables have low information entropy and nonhomogenous variables have high information entropy. This was done in the calculation of $I_{\text{outlook,no partition}}$.
2. Weight the influence of each independent variable on the target or dependent variable using the entropy weighted averages (sometimes called joint entropy). This was done during the calculation of $I_{\text{outlook}}$ in the above example.
3. Compute the information gain, which is essentially the reduction in the entropy of the target variable due to its relationship with each independent variable. This is simply the difference between the information entropy found in step 1 minus the joint entropy calculated in step 2. This was done during the calculation of $I_{\text{outlook,no partition}} - I_{\text{outlook}}$.
4. The independent variable with the highest information gain will become the "root" or the first node on which the data set is divided. This was done during the calculation of the information gain table.
5. Repeat this process for each variable for which the Shannon entropy is nonzero. If the entropy of a variable is zero, then that variable becomes a "leaf" node.

### 4.1.2 How to Implement

Before jumping into a business use case of decision trees, let us "implement" the decision tree model that was shown in Figure 4.4 on a small test sample using RapidMiner. Figure 4.5 shows the test data set, which is very much like our training data set but with small differences in attribute values.

We have built a decision tree model using training data set. The RapidMiner process of building a decision tree is shown in Figure 4.6. More over, the same process shows the application of the decision tree model to test data set. When this process is executed, and the connections are made as shown in

/iew ExampleSet (Retrieve Golf-Testset) ExampleSet (Retriev

ExampleSet (14 examples, 1 special attribute, 4 regular attributes)

| Row No. | Play | Outlook | Temperature | Humidity | Wind |
|---|---|---|---|---|---|
| 1 | yes | sunny | 85 | 85 | false |
| 2 | no | overcast | 80 | 90 | true |
| 3 | yes | overcast | 83 | 78 | false |
| 4 | yes | rain | 70 | 96 | false |
| 5 | yes | rain | 68 | 80 | true |
| 6 | no | rain | 65 | 70 | true |
| 7 | yes | overcast | 64 | 65 | true |
| 8 | no | sunny | 72 | 95 | false |
| 9 | yes | sunny | 69 | 70 | false |
| 10 | no | sunny | 75 | 80 | false |
| 11 | yes | sunny | 68 | 70 | true |
| 12 | yes | overcast | 72 | 90 | true |
| 13 | no | overcast | 81 | 75 | true |
| 14 | yes | rain | 71 | 80 | true |

**FIGURE 4.5**
Golf: Test data has a few minor differences in attribute values from the training data.

Figure 4.6, we get the table output shown in Figure 4.7. You can see that the model has been able to get 9 of the 14 class predictions correct and 5 of the 14 (in boxes) wrong, which translates to about 64% accuracy.

Let us examine a more involved business application to better understand how to apply decision trees for real world problems. Credit scoring is a fairly common predictive analytics problem. Some types of situations where credit scoring could be applied are:

1. **Prospect filtering**: Identify which prospects to extend credit to and determine how much credit would be an acceptable risk.
2. **Default risk detection**: Decide if a particular customer is likely to default on a loan.
3. **Bad debt collection**: Sort out those debtors who will yield a good cost (of collection) to benefit (of receiving payment) performance.

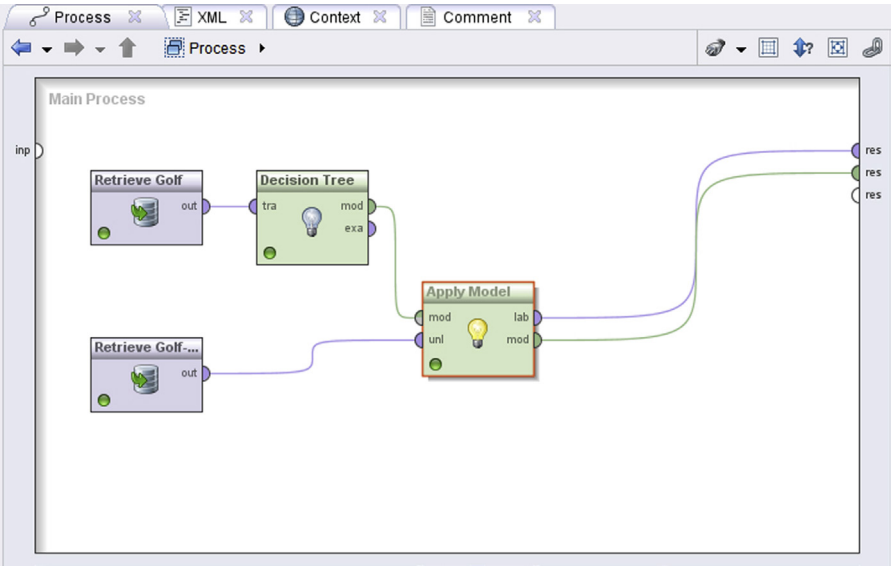



**FIGURE 4.6**
Applying the simple decision tree model on unseen golf test data.

We will use the well-known German Credit data set from the University of California-Irvine Machine Learning data repository[1] and describe how to use RapidMiner to build a decision tree for addressing a *prospect filtering problem*.

This is the first discussion of the implementation of a predictive analytics technique, so we will spend some extra effort in going into detail on many of the preliminary steps and also introduce several additional tools and concepts that will be required throughout the rest of this chapter and other chapters that focus on supervised learning methods. These are the concepts of *splitting data into testing and training samples*, and *applying the trained model* on testing (or validation data). It may also be useful to first review Sections 13.1 (Introduction to the GUI) and 13.2 (Data Import and Export) from Chapter 13 Getting Started with RapidMiner before working through the rest of this implementation. As a final note, we will not be discussing ways and means to improve the performance of a classification model using RapidMiner in this section, but will return to this very important part of predictive analytics in several later chapters, particularly in the section on using optimization in Chapter 13.

[1]http://archive.ics.uci.edu/ml/datasets/Statlog+%28German+Credit+Data%29. All data sets used in this book are available at the companion website.

ExampleSet (14 examples, 4 special attributes,

| Row No. | Play | prediction(Play) |
|---|---|---|
| 1 | yes | no |
| 2 | no | yes |
| 3 | yes | yes |
| 4 | yes | yes |
| 5 | yes | no |
| 6 | no | no |
| 7 | yes | yes |
| 8 | no | no |
| 9 | yes | yes |
| 10 | no | no |
| 11 | yes | yes |
| 12 | yes | yes |
| 13 | no | yes |
| 14 | yes | no |

**FIGURE 4.7**
Results of applying the simple decision tree model.

There are four main steps in setting up any supervised learning algorithm for a predictive modeling exercise:

1. Read in the cleaned and prepared data (see Chapter 2 Data Mining Process), typically from a spreadsheet, but the data can be from any source.
2. Split data into training and testing samples.
3. Train the decision tree using the training portion of the data set.
4. Apply the model on the testing portion of the data set to evaluate the performance of the model.

Step 1 may seem rather elementary, but can confuse many beginners and thus we will spend some time explaining this in somewhat more detail. The next few parts will describe other steps also in detail.

### *Step 1: Data Preparation*

The raw data is in the format shown in Table 4.3. It consists of 1,000 samples and a total of 20 attributes and 1 label or target attribute. There are seven numeric attributes and the rest are categorical or qualitative, including the

**Table 4.3** A View of the Raw German Credit data.

| Checking Account Status | Duration in Month | Credit History | Purpose | Credit Amount | Savings Account/ Bonds | Present Employment Since | Credit Rating |
|---|---|---|---|---|---|---|---|
| A11 | 6 | A34 | A43 | 1169 | A65 | A75 | 1 |
| A12 | 48 | A32 | A43 | 5951 | A61 | A73 | 2 |
| A14 | 12 | A34 | A46 | 2096 | A61 | A74 | 1 |
| A11 | 42 | A32 | A42 | 7882 | A61 | A74 | 1 |
| A11 | 24 | A33 | A40 | 4870 | A61 | A73 | 2 |
| A14 | 36 | A32 | A46 | 9055 | A65 | A73 | 1 |
| A14 | 24 | A32 | A42 | 2835 | A63 | A75 | 1 |
| A12 | 36 | A32 | A41 | 6948 | A61 | A73 | 1 |
| A14 | 12 | A32 | A43 | 3059 | A64 | A74 | 1 |
| A12 | 30 | A34 | A40 | 5234 | A61 | A71 | 2 |
| A12 | 12 | A32 | A40 | 1295 | A61 | A72 | 2 |
| A11 | 48 | A32 | A49 | 4308 | A61 | A72 | 2 |

label, which is a binomial variable. The label attribute is called Credit Rating and can take the value of 1 (good) or 2 (bad). In the data 70% of the samples fall into the "good" credit rating class. The descriptions for the data are shown in Table 4.3. Most of the attributes are self-explanatory, but the raw data has encodings for the values of the qualitative variables. For example, attribute 4 is the *purpose of the loan* and can assume any of 10 values (A40 for new car, A41 for used car, and so on). The full details of these encodings are provided under the "Data Set Description" on the UCI-ML website.

RapidMiner's easy interface allows quick importing of spreadsheets. A useful feature of the interface is the panel on the left, called the "Operators." Simply typing in text in the box provided automatically pulls up all available RapidMiner operators that match the text. In this case, we need an operator to read an Excel spreadsheet, and so we simply type "excel" in the box. As you can see, the three Excel operators are immediately shown in Figure 4.8a: two for reading and one for exporting data.

Either double-click on the *Read Excel* operator or drag and drop it into the Main Process panel—the effect is the same, see Figure 4.8b. Once the *Read Excel* operator appears in the main process window, we need to configure the data import process. What this means is telling RapidMiner which columns to import, what is contained in the columns, and if any of the columns need special treatment.

This is probably the most "cumbersome" part about this step. RapidMiner has a feature to automatically detect the type of values in each attribute (Guess Value types). But it is a good exercise for the analyst to make sure that the right

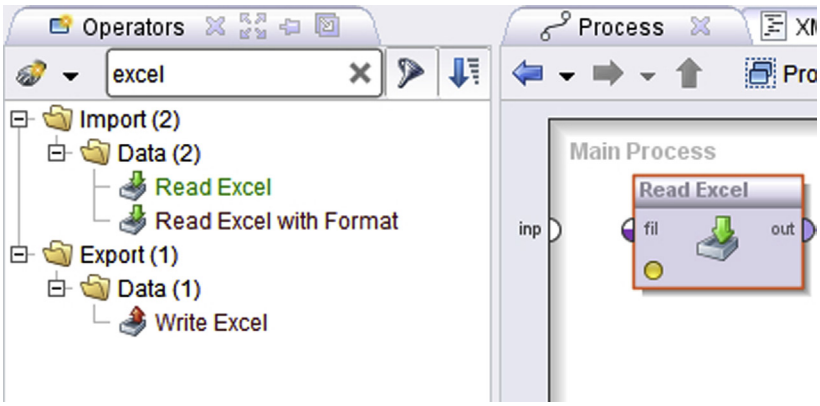



**FIGURE 4.8a**
Using the Read Excel operator.

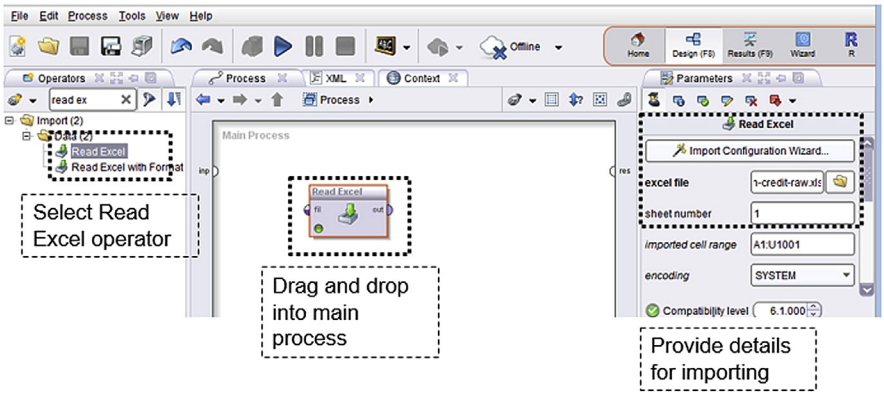



**FIGURE 4.8b**
Configuring the Read Excel operator.

columns are picked (or excluded) and the value types are correctly guessed. If not, as seen in Figure 4.9, we can change the value type to the correct setting by clicking on the button below the attribute name.

Once the data is imported, we must assign the target variable for analysis, also known as a "label." In this case, it is the Credit Rating, as shown in Figure 4.9. Finally it is a good idea to "run" RapidMiner and generate results to ensure that all columns are read correctly.

An optional step is to convert the values from A121, A143, etc. to more meaningful qualitative descriptions. This is accomplished by the use of another operator called *Replace (Dictionary)*, which will replace the values with bland encodings such as A121 and so on with more descriptive values. We need to create a dictionary and supply this to RapidMiner as a comma-separated value (csv) file to enable this. Such a dictionary is easy to create and is shown in

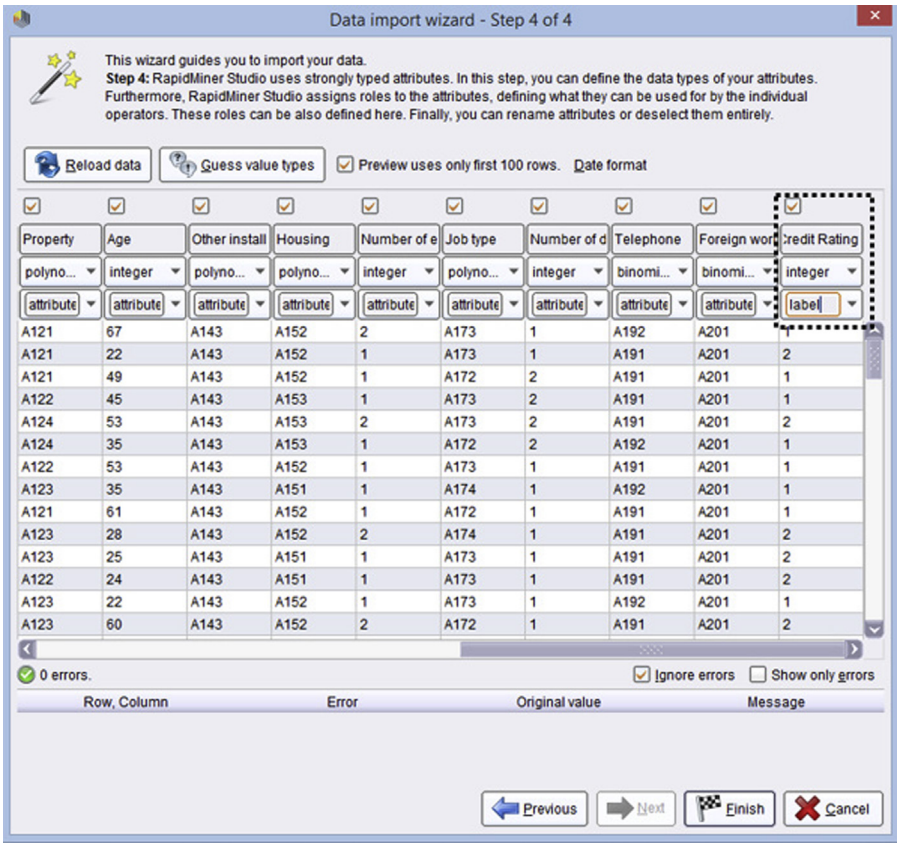



**FIGURE 4.9**
Verifying data read-in and adjusting attribute value types if necessary.

Figure 4.10a; the setup is shown in Figure 4.10b. Note that we need to let RapidMiner know which column in our dictionary contains old values and which contains new values.

The last preprocessing step we show here is converting the numeric label (see Figure 4.9) into a binomial one by connecting the "exa"mple output of *Replace (Dictionary)* to a *Numerical to Binominal* operator. To configure the *Numerical to Binominal* operator, follow the setup shown in Figure 4.10c.

Finally, let us change the name of the label variable from Credit Rating to Credit Rating = Good so that it makes more sense when the integer values get converted to true or false after passing through the *Numerical to Binominal* operator. This can be done using the *Rename* operator. When we run this setup, we will generate the data set shown in Figure 4.11. Comparing to Figure 4.9, we will see that the label attribute is the first one shown and the values are "true" or "false." We can examine the statistics tab of the results to get more information about the distributions of individual attributes and also to check for

File Edit Format View Help

```
oldValue,newValue
A11,"Less than 0 DM"
A12,"0 to 200 DM"
A13,"Greater than 200 DM"
A14,"no checking account"
A30,"no credits taken all credits paid back duly"
A31,"all credits at this bank paid back duly"
A32,"existing credits paid back duly till now"
A33,"delay in paying off in the past"
A34,"critical account other credits existing (not at this bank)"
A40,"new car"
A41,"used car"
A42,"furniture equipment"
A43,"radio television"
A44,"domestic appliances"
A45,"repairs"
A46,"education"
A47,"vacation"
A48,"retraining"
A49,"business"
A410,"others"
A61,"Less than 100 DM"
A62,"100 to 500 DM"
A63."500 to 1000 DM"
```

**FIGURE 4.10a**
Attribute value replacement using a dictionary.

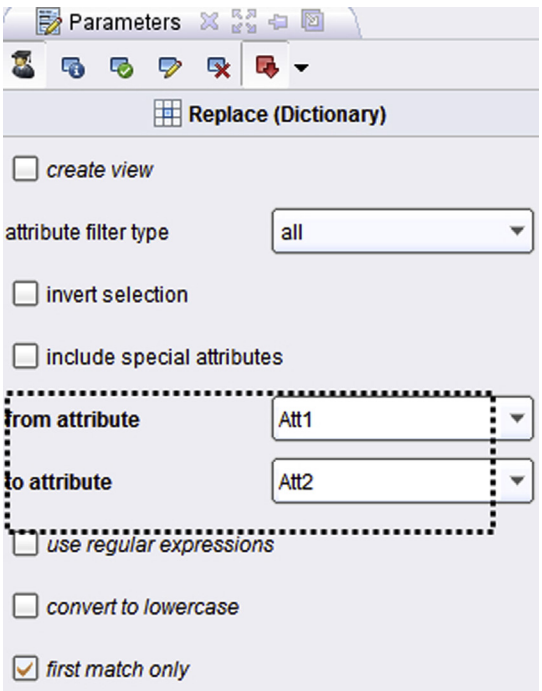



**FIGURE 4.10b**
Configuring the Replace (Dictionary) operator.

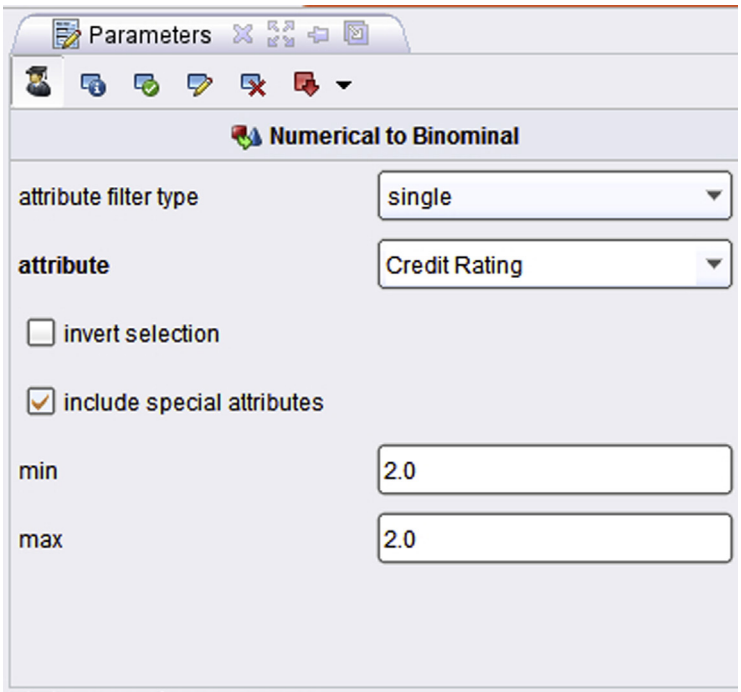



**FIGURE 4.10c**
Convert the integer Credit Rating label variable to a binomial (true or false) type.

missing values and outliers. In other words, we must make sure that the data preparation step (Section 2.2) is properly executed before proceeding. In this implementation, we will not worry about this because the data set is relatively "clean" (for instance, there are no missing values), and we can proceed directly to the model development phase.

### *Step 2: Divide Data Set into Training and Testing Samples*

As with all supervised model building, data must be separated into two sets: one for "training" or developing an acceptable model, and the other for "validating" or ensuring that the model would work equally well on a different data set. The standard practice is to split the available data into a training set and a testing or validation set. Typically the training set contains 70% to 90% of the original data. The remainder is set aside for testing or validation.

Figure 4.12 shows how to do this in RapidMiner. The *Split Validation* tool sets up splitting, modeling, *and* the validation check in one operator. The utility of this will become very obvious as you develop experience in data mining, but as a beginner, this may be a bit confusing.

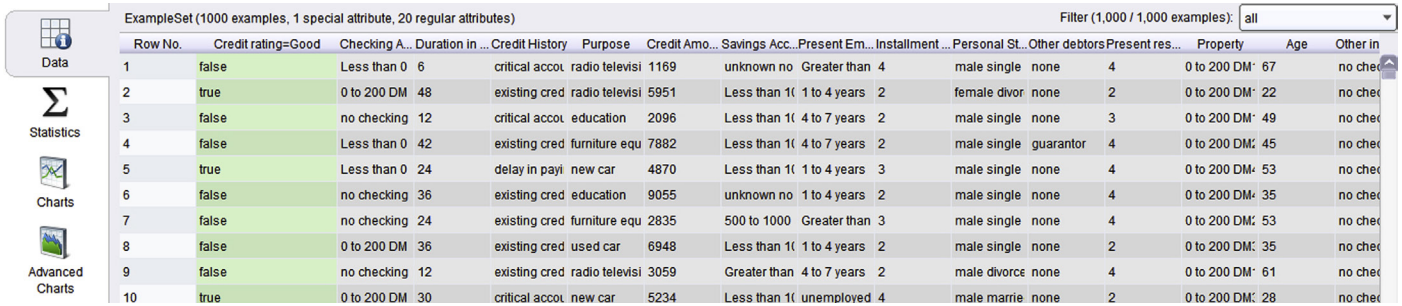



**FIGURE 4.11**

Data transformed for decision tree analysis.

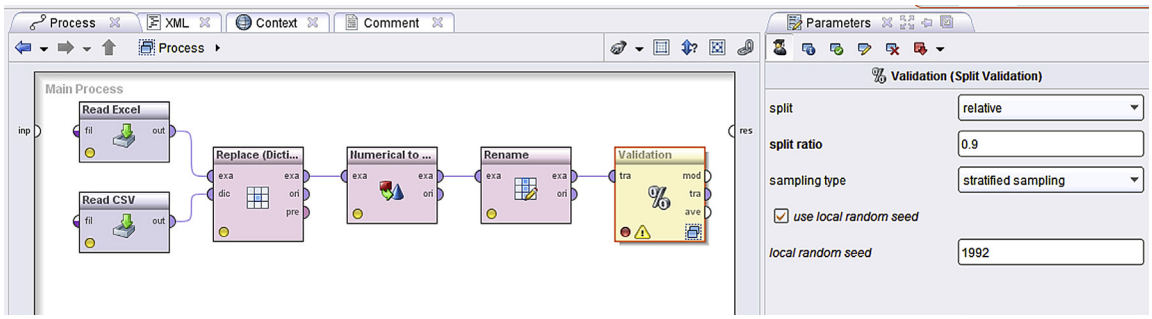



**FIGURE 4.12**
Using a *relative* split and a split ratio of 0.9 for training versus testing.

Choose *stratified sampling* with a split ratio of 0.9 (90% training). Stratified sampling will ensure that both training and testing samples have equal distributions of class values. (Although not necessary, it is sometimes useful to check the *use local random seed* option, so that it is possible to compare models between different iterations. Fixing the random seed ensures that the same examples are chosen for training (and testing) subsets each time the process is run.) The final sub step here is to connect the output from the *Numerical to Binominal* operator output to the *Split Validation* operator input.[2]

### *Step 3: Modeling Operator and Parameters*

We will now see how to build a decision tree model on this data. As mentioned earlier, the *Validation* operator allows us to build a model and apply it on validation data in the same step. This means that two operations—model building and model evaluation—must be configured using the same operator. This is accomplished by double-clicking on the *Validation* operator, which is what is called a "nested" operator. All nested operators in RapidMiner have two little blue overlapping windows on the bottom right corner. When this operator is "opened," we can see that there are two parts inside (see Figure 4.13). The left box is where the *Decision Tree* operator has to be placed and the model will be built using the 90% of training data samples. The right box is for applying this trained model on the remaining 10% of the testing data samples using the *Apply Model* operator and evaluating the performance of the model using the *Performance* operator.

### *Configuring the Decision Tree Model*

The main parameters to pay attention to are the Criterion pull-down menu and the minimal gain box. This is essentially a partitioning criterion and offers information gain, Gini index, and gain ratio as choices. We covered the first two criteria earlier, and the gain ratio will be briefly explained in the next section.

[2]This is just a basic sampling technique and the sampling itself can involve a lot of work to validate and produce the correct sampling.

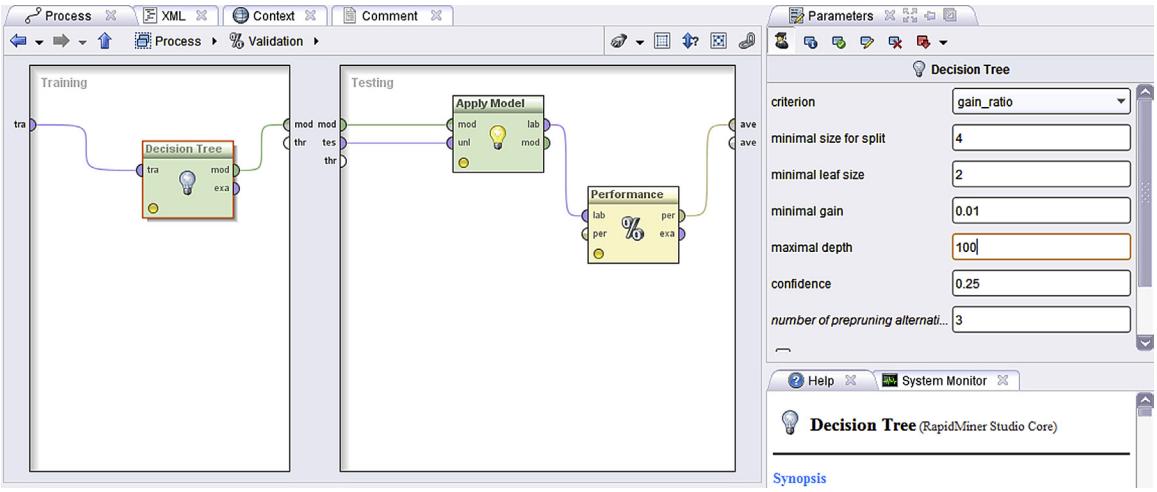



**FIGURE 4.13**
Setting up the split validation process.

As discussed earlier in this chapter, decision trees are built up in a simple five-step process by increasing the information contained in the reduced data set following each split. Data by its nature contains uncertainties. We may be able to systematically reduce uncertainties and thus increase information by activities like sorting or classifying. When we have sorted or classified to achieve the greatest reduction in uncertainty, we have basically achieved the greatest increase in information. We have seen how entropy is a good measure of uncertainty and how keeping track of it allows us to quantify information. So this brings us back to the options that are available within RapidMiner for splitting decision trees:

1. **Information gain:** Simply put, this is computed as the information before the split minus the information after the split. It works fine for most cases, unless you have a few variables that have a large number of values (or classes). Information gain is *biased* towards choosing attributes with a large number of values as root nodes. This is not a problem, except in extreme cases. For example, each customer ID is unique and thus the variable has too many values (each ID is a unique value). A tree that is split along these lines has no predictive value.
2. **Gain ratio (default):** This is a modification of information gain that reduces its bias and is usually the best option. Gain ratio overcomes the problem with information gain by taking into account the number of branches that would result before making the split. It corrects information gain by taking the *intrinsic information* of a split into account. Intrinsic information can be explained using our golf example.

Suppose each of the 14 examples had a unique ID attribute associated with them. Then the intrinsic information for the ID attribute is given by 14 * (–1/14 * log (1/14)) = 3.807. The gain ratio is obtained by dividing the information gain for an attribute by its intrinsic information. Clearly attributes that have very high intrinsic information (high uncertainty) tend to offer low gains upon splitting and hence will not be automatically selected.

3. **Gini index:** This is also used sometimes, but does not have too many advantages over gain ratio.
4. **Accuracy:** This is also used to improve performance. The best way to select values for these parameters is by using many of the optimizing operators. This is a topic that will be covered in detail in Chapter 13.

The other important parameter is the *minimal gain* value. Theoretically this can take any range from 0 upwards. In practice, a minimal gain of 0.2 to 0.3 is considered good. The default is 0.1.

The other parameters (*minimal size for a split*, *minimal leaf size*, *maximal depth*) are determined by the size of the data set. In this case, we proceed with the default values. The best way to set these parameters is by using an optimization routine (which will be briefly introduced in Chapter 13 Getting Started with RapidMiner.

The last step in training the decision tree is to connect the input ports ("tra"in-ing) and output ports ("mod"el) as shown in the left (training) box of Figure 4.9. The model is ready for training. Next we add two more operators, *Apply Model* and *Performance (Binominal Classification)*, and we are ready to run the analysis. Configure the *Performance (Binominal Classification)* operator by selecting the *accuracy*, *AUC*, *precision*, and *recall* options.[3]

Remember to connect the ports correctly as this can be a source of confusion:

- "mod"el port of the Testing window to "mod" on Apply Model
- "tes"ting port of the Testing window to "unl"abeled on Apply Model
- "lab"eled port of Apply Model to "lab"eled on Performance
- "per"formance port on the *Performance* operator to "ave"rageable port on the output side of the testing box

The final step before running the model is to go back to the main perspective by clicking on the blue up arrow on the top left (see Figure 4.13) and connect the output ports "mod"el and "ave" of the *Validation* operator to the main outputs.

### *Step 4: Process Execution and Interpretation*

When the model is run as setup above, RapidMiner generates two tabs in the Results perspective (refer to Chapter 13 for the terminology).

[3]Performance criteria such as these are explained in Chapter 8 on evaluation.

The *PerformanceVector (Performance)* tab shows a confusion matrix that lists the model accuracy on the testing data, along with the other options selected above for the *Performance (Binominal Classification)* operator in step 3. The Tree (Decision Tree) tab shows a graphic of the tree that was built on the training data (see Figure 4.14). Several important points must be highlighted before we discuss the performance of this model:

1. The root node—Checking Account Status—manages to classify nearly 94% of the data set. This can be verified by hovering the mouse over each of the three terminal leaves for this node. The total occurrences (of Credit Rating = Good: *true* and *false*) for this node alone are 937 out of 1000. In particular, if someone has a *Checking Account Status = no checking account*, then the chances of them having a *"true"* score is 88% (= 348/394, see Figure 4.15).
2. However, the tree is unable to clearly pick out true or false cases for Credit Rating = Good if *Checking Account Status* is *Less than 0 DM*

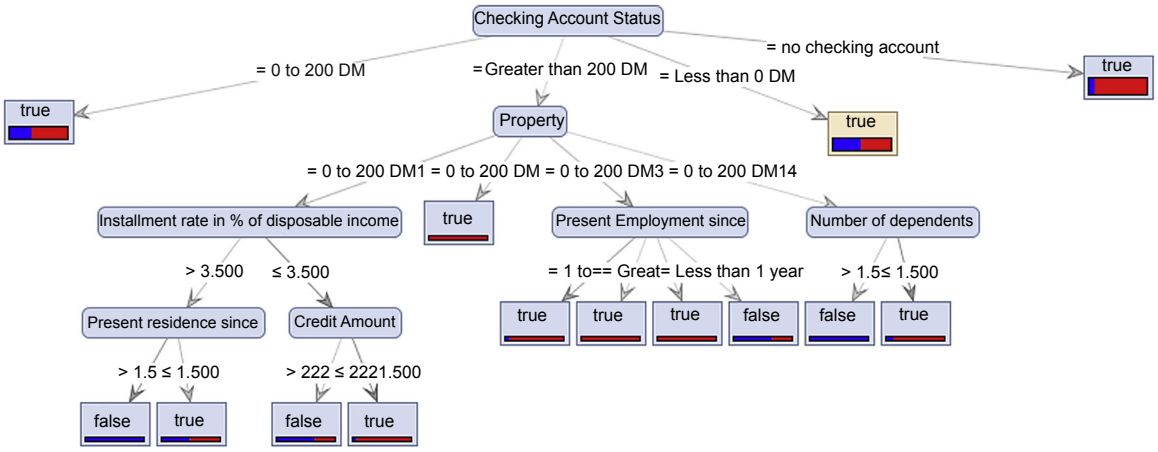



**FIGURE 4.14**
A decision tree model for the prospect scoring data.

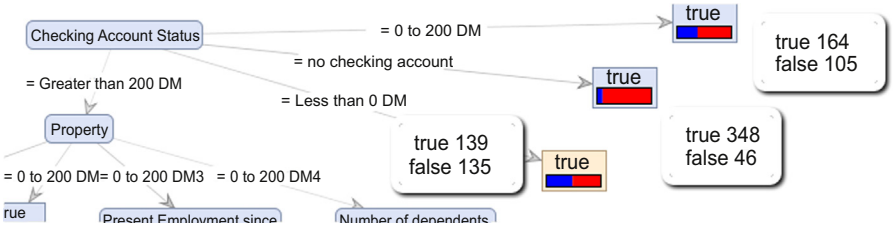



**FIGURE 4.15**
Predictive power of the root node of the decision tree model.

*(or Deutsche mark)* (only a 51% chance of correct identification). A similar conclusion results if someone has *0 to 200 DM*.

3. If the *Checking Account Status* is greater than 200 DM, then the other parameters come into effect and play an increasingly important role in deciding if someone is likely to have a *"good"* or *"bad"* credit rating.
4. However, the fact that there are numerous terminal leaves with frequencies of occurrence as low as 2 (for example, "Present Employment since"), it implies that the tree suffers from overfitting. As described earlier, overfitting refers to the process of building a model very specific to the training data that achieves close to full accuracy on the training data. However when this model is applied on new data or if the training data changes somewhat, then there is a significant degradation in its performance. Overfitting is a potential issue with all supervised models, not just decision trees. One way we could have avoided this situation is by changing the decision tree criterion *"Minimal leaf size"* to something like 10 (instead of the default, 2). But doing so, we would also lose the classification influence of all the other parameters, except the root node [try it!]

Now let us look at the Performance result. As seen in Figure 4.16, the model's overall accuracy on the testing data is 72%. The model has a class recall of 100% for the "true" class implying that it is able to pick out customers with good credit rating with 100% accuracy. However, its class recall for the "false" class is an abysmal 6.67%! That is, the model can only pick out a potential defaulter in 1 out of 15 cases!

One way to improve this performance is by penalizing false negatives by applying a cost for every such instance. This is handled by another operator called *MetaCost*, which is described in detail in Chapter 5 in the section on logistic regression. When we perform a parameter search optimization by iterating through three of the decision tree parameters, splitting criterion, minimum gain ratio, and maximal tree depth, we hit upon significantly improved performance as seen in Figure 4.17 below. More details on how to set this type of optimization are provided in Chapter 13.

When we run the best model (described by the parameters on the top row of the table in Figure 4.17), we obtain the confusion matrix shown in Figure 4.18.

accuracy: 72.00%

| | true false | true true | class precision |
|---|---|---|---|
| pred. false | 2 | 0 | 100.00% |
| pred. true | 28 | 70 | 71.43% |
| class recall | 6.67% | 100.00% | |

**FIGURE 4.16**
Baseline model performance measures.

| criterion | gain | tree depth | accuracy | recall |
|---|---|---|---|---|
| gain_ratio | 0.010 | 12 | 0.780 | 0.914 |
| gini_index | 0.010 | 5 | 0.760 | 0.871 |
| gini_index | 0.039 | 5 | 0.760 | 0.871 |
| gini_index | 0.068 | 5 | 0.760 | 0.871 |
| gini_index | 0.097 | 5 | 0.760 | 0.871 |
| gini_index | 0.126 | 5 | 0.760 | 0.871 |
| gini_index | 0.155 | 5 | 0.760 | 0.871 |
| gini_index | 0.184 | 5 | 0.760 | 0.871 |
| gini_index | 0.213 | 5 | 0.760 | 0.871 |
| gini_index | 0.242 | 5 | 0.760 | 0.871 |
| gini_index | 0.271 | 5 | 0.760 | 0.871 |
| gini_index | 0.300 | 5 | 0.760 | 0.871 |
| gini_index | 0.010 | 18 | 0.760 | 0.857 |
| gini_index | 0.039 | 18 | 0.760 | 0.857 |

**FIGURE 4.17**
Optimizing the decision tree parameters to improve accuracy and class recall.

| accuracy: 78.00% | | | |
|---|---|---|---|
| | true false | true true | class precision |
| pred. false | 64 | 16 | 80.00% |
| pred. true | 6 | 14 | 70.00% |
| class recall | 91.43% | 46.67% | |

**FIGURE 4.18**
Optimizing class recall for the credit default identification process.

Comparing this to Figure 4.16 we see that the recall for the more critical class (correctly identifying cases with a bad credit rating), has increased from about 7% to 91% whereas the recall for identifying a good credit rating has fallen below 50%. This may be acceptable in this particular situation if the costs of issuing a loan to a potential defaulter are significantly higher than the costs of losing revenue by denying credit to a creditworthy customer. The overall accuracy of the model is also higher than before.

In addition to assessing the model's performance by aggregate measures such as accuracy, we can also use gain/lift charts, receiver operator characteristic (ROC) charts, and area under ROC curve (AUC) charts. An explanation of how these charts are constructed and interpreted is given in Chapter 8 on model evaluation.

The RapidMiner process for a decision tree covered in the implementation section can be accessed from the companion site of the book at www.LearnPredictiveAnalytics.com. The RapidMiner process (*.rmp files) can be downloaded to the computer and imported to RapidMiner through File → Import Process. Additionally, all the data sets used in this book can be downloaded from www.LearnPredictiveAnalytics.com

### 4.1.3 Conclusions

Decision trees are one of the most commonly used predictive modeling algorithms in practice. The reasons for this are many. Some of the distinct advantages of using decision trees in many classification and prediction applications are explained below along with some common pitfalls.

- Easy to interpret and explain to nontechnical users

As we have seen in the few examples discussed so far, decision trees are very intuitive and easy to explain to nontechnical people, who are typically the consumers of analytics.

- Decision trees require relatively little effort from users for data preparation

There are several points that add to the overall advantages of using decision trees. If we have a data set consisting of widely ranging attributes, for example, revenues recorded in millions and loan age recorded in years, many algorithms require scale normalization before model building and application. Such variable transformations are not required with decision trees because the tree structure will remain the same with or without the transformation.

Another feature that saves data preparation time: missing values in training data will not impede partitioning the data for building trees. Decision trees are also not sensitive to outliers since the partitioning happens based on the proportion of samples within the split ranges and not on absolute values.

- Nonlinear relationships between parameters do not affect tree performance

As we describe in Chapter 5 on linear regression, highly nonlinear relationships between variables will result in failing checks for simple regression models and thus make such models invalid. However, decision trees do not require any assumptions of linearity in the data. Thus, we can use them in scenarios where we know the parameters are nonlinearly related.

- Decision trees implicitly perform variable screening or feature selection

We will discuss in Chapter 12 why feature selection is important in predictive modeling and data mining. We will introduce a few common techniques for performing feature selection or variable screening in that chapter. But when we fit a decision tree to a training data set, the top few nodes on which the tree is split are essentially the most important variables within the data set and feature selection is completed automatically. In fact, RapidMiner has an operator for performing variable screening or feature selection using the information gain ratio.

However, all these advantages need to be tempered with one key disadvantage of decision trees: without proper pruning or limiting tree growth, they tend to overfit the training data, making them somewhat poor predictors.

## 4.2 RULE INDUCTION

Rule induction is a data mining process of deducing if-then rules from a data set. These symbolic decision rules explain an inherent relationship between the attributes and class labels in the data set. Many real-life experiences are based on intuitive rule induction. For example, we can proclaim a rule that states "if it is 8 a.m. on a weekday, then highway traffic will be heavy" and "if it is 8 p.m. on a Sunday, then the traffic will be light." These rules are not necessarily right all the time. 8 a.m. weekday traffic may be light during a holiday season. But, in general, these rules hold true and are deduced from real-life experience based on our every day observations. Rule induction provides a powerful classification approach that can be easily understood by the general audience. Apart from its use in Predictive Analytics by classification of unknown data, rule induction is also used to describe the patterns in the data. The description is in the form of simple if-then rules that can be easily understood by general users.

The easiest way to extract rules from a data set is from a decision tree that is developed on the same data set. A decision tree splits data on every node and leads to the leaf where the class is identified. If we trace back from the leaf to the root node, we can combine all the split conditions to form a distinct rule. For example, in the Golf data set (Table 4.1), based on four weather conditions, we can generalize a rule set to determine when a player prefers to play golf or not. As discussed in the decision tree section, a decision tree from Golf data set can be developed. Figure 4.19 shows the decision tree developed from the Golf data with five leaf nodes and two levels. If we trace back the first leaf from the left, we can extract a rule: *If Outlook is overcast, then Play = yes*. Similarly, rules can be extracted from all the five leaves:

Rule 1: if (Outlook = overcast) then Play = yes
Rule 2: if (Outlook = rain) and (Wind = false) then Play = yes
Rule 3: if (Outlook = rain) and (Wind = true) then Play = no

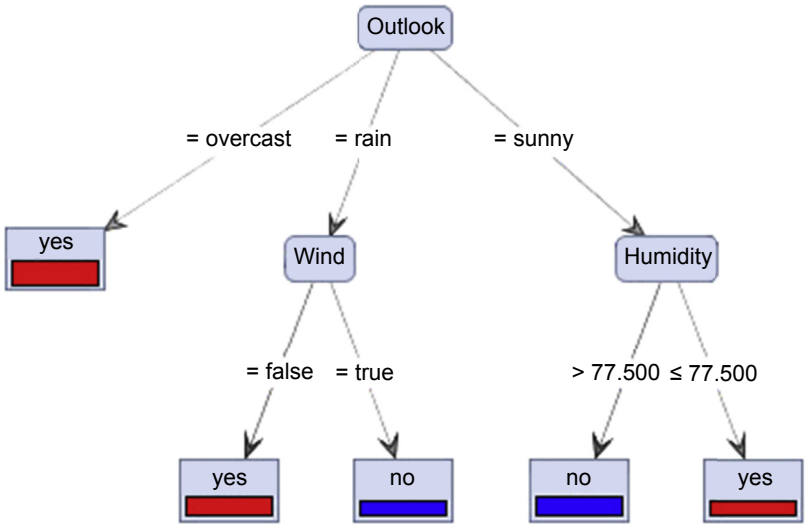



**FIGURE 4.19**
Approaches to rule generation.

Rule 4: if (Outlook = sunny) and (Humidity > 77.5) then Play = no
Rule 5: if (Outlook = sunny) and (Humidity ≤ 77.5) then Play = yes

The set of all the five rules is called a *rule set*. Each individual rule $r_i$ is called a *disjunct* or classification rule. The entire rule set can be represented as

$$R = \{r_1 \cap r_2 \cap r_3 \cap .. r_k\}$$

where k is the number of disjuncts in a rule set. Individual disjuncts can be represented as

$$r_i = (\text{antecedent or condition}) \text{ then } (\text{consequent})$$

For example, rule $r_2$: if (Outlook = rain) and (Wind = false) then Play = yes.

In the above example rule, (Outlook = rain) and (Wind = false) is the *antecedent* or *condition* of the rule. The condition of the rule can have many attributes and values each separated by a logical AND operator. Each attribute and value test is called the *conjunct* of the rule. An example of a conjunct is (Outlook = rain). The antecedent is a group of conjuncts with the AND operator. Each conjunct is a node in the equivalent decision tree.

In the Golf data set, we can observe a couple of properties of the rule set in relation with the data set. First, the rule set is mutually exclusive. This means that no example record will trigger more than one rule and hence the outcome of the prediction is definite. However, there can be rule sets that are not mutually exclusive. If a record activates more than one rule in a

rule set and all the class predictions are the same, then there is no problem. If the class predictions differ, ambiguity exists on which class is the prediction of the induction rule model. There are a couple of techniques used to resolve conflicting class prediction by more than one rule. One technique is to develop an ordered list of rules where if a record activates many rules, the first rule in the order will take precedence. A second technique is where each active rule can "vote" for a prediction class. The predicted class with highest vote is the prediction of the rule set. The rule set discussed is also exhaustive. This means the rule set is activated for all the combinations of the attribute values in the record set, not just limited to training records. If the rule set is not exhaustive, then a final catch all bucket rule *"else Class = Default Class Value"* can be introduced to make the rule set exhaustive.

### 4.2.1 Approaches to Developing a Rule Set

Rules can be directly extracted from the data set or derived from the previously built decision trees from the same data set. Figure 4.20 shows the approaches to generate rules from the data set. The former approach is called the *direct* method, which is built on leveraging the relationship between the attribute and class label in the data set. Deriving a rule set from a previously built classifier decision tree model is a passive or *indirect* approach. Since building a decision tree is covered in previous section and the derivation of rules from the decision tree model is straightforward, we will focus the rest of the discussion on direct rule generation based on the relationship from the data. Specifically, we will focus on the *sequential covering* technique to build a rule set.

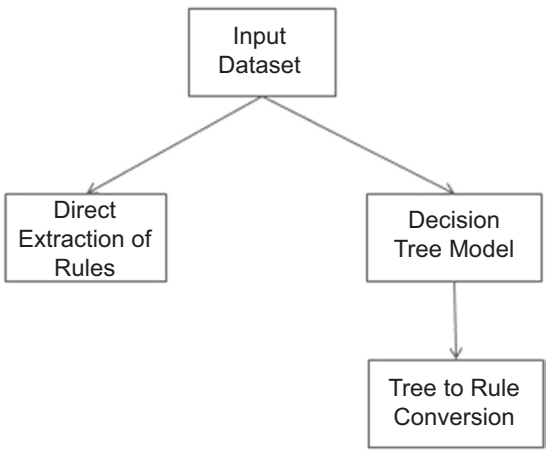



**FIGURE 4.20**
Decision tree model for Golf data set.

## PREDICTING AND PREVENTING MACHINE BREAKDOWNS

A machine breakdown in the field almost always results in disruption of a manufacturing process. In a large-scale process like an oil refinery, chemical plants, etc., it causes serious financial damage to the company and manufacturers of the machines. Let's assume the machine under consideration is a motor. Rather than waiting for the machine to break down and react, it is much preferable to diagnose the problem and prevent the breakdown before a problem occurs. Large-scale machines track thousands of real-time readings from multiple parts of the machine (Such machines connected to networks that can gather readings and act based on smart logic are called Internet of Things (IoT).). One of the solutions is to leverage how these readings are trending and develop a rule base which says, for example, *if the cylinder temperature continues to report more than 852°C, then the machine will break down in the near future*. These types of the rules are simple to interpret, don't require an expert to be around to take further action, and can be deployed by automated systems.

Developing learned rules requires historical analysis of all the readings that lead up to a machine failure (Langley & Simon, 1995). These learned rules are different and in many cases supersede the rule of thumb assumed by the machine expert. Based on the historic readings of a failure event and nonfailure events, the learned rule set can predict the failure of the machine and hence can alert the operator of imminent future breakdowns. Since these rules are very simple to understand, these preventive measures can be easily deployed to line workers. This use case demonstrates the need of not only a predictive data model, but also a descriptive model where the inner working of the model can be easily understood by the users. A similar approach can be developed to prevent customer churn, or loan default, for example.

### 4.2.2 How it Works

Sequential covering is an iterative procedure of extracting rules from the data set. The sequential covering approach attempts to find all the rules in the data set class by class. One specific implementation of the sequential covering approach is called the RIPPER, which stands for Repeated Incremental Pruning to Produce Error Reduction (Cohen, 1995). Consider the data set shown in Figure 4.21, which has two dimensions on the X and Y axis and two class labels marked by "+" and "−". Following are the steps in sequential covering rules generation approach (Tan et al. 2005).

#### *Step 1: Class Selection*

The algorithm starts with selection of class labels one by one. The rule set is class-ordered where all the rules for a class are developed before moving on to next class. The first class is usually the least-frequent class label. From Figure 4.21, the least frequent class is "+" and the algorithm focuses on generating all the rules for "+" class.

#### *Step 2: Rule Development*

The objective in this step is to cover all "+" data points using classification rules with none or as few "−" as possible. For example, in Figure 4.22, rule $r_1$ identifies the area of four "+" in the top left corner. Since this rule is based on simple logic operators in conjuncts, the boundary is rectilinear.

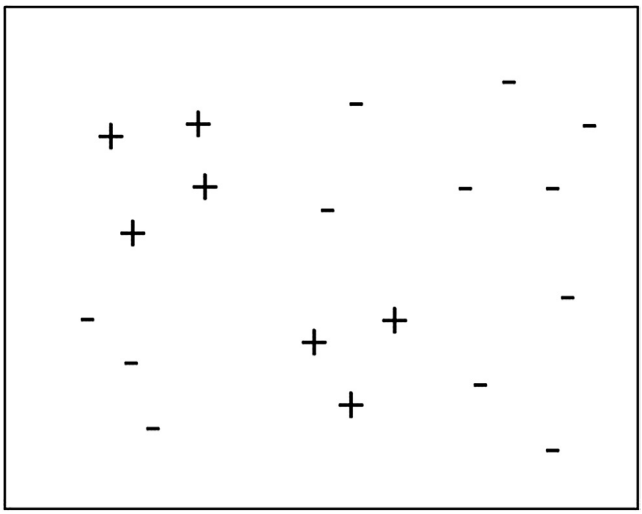


**FIGURE 4.21**
Data set with two classes and two dimensions.

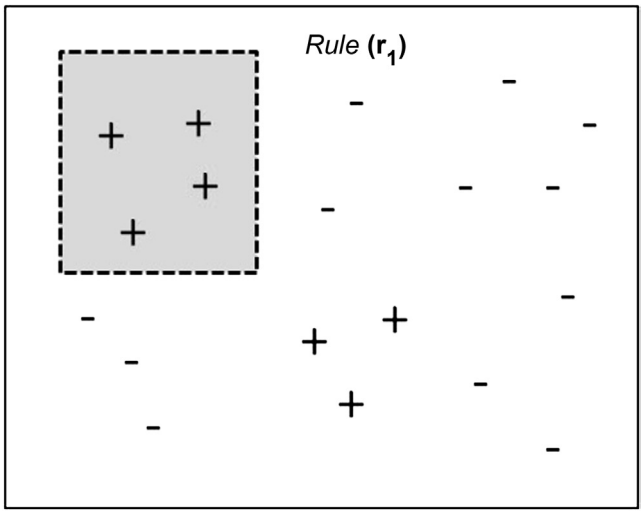



**FIGURE 4.22**
Generation of rule $r_1$.

Once rule $r_1$ is formed, the entire data points covered by $r_1$ are eliminated and the next best rule is found from data sets. The algorithm grows in a greedy fashion using a technique called Learn-One-Rule which is described in the next section. One of the outcomes of greedy algorithms that start

with initial configuration is that they yield *local optima* instead of a *global optimum*. A local optimum is a solution that is optimal in the neighborhood of potential solutions, but worse than the global optimum.

### *Step 3: Learn-One-Rule*

Each rule $r_i$ is grown by the learn-one-rule approach. Each rule starts with an empty rule set and conjuncts are added one by one to increase the rule accuracy. Rule accuracy is the ratio of amount of + covered by the rule to all records covered by the rule:

$$\text{Rule accuracy } A(r_i) = \frac{\text{Correct records covered by rule}}{\text{All records covered by the rule}}$$

Learn-one-rule starts with an empty rule set: if {} then class = "+". Obviously the accuracy of this rule is the same as the proportion of + data points in the data set. Then the algorithm greedily adds conjuncts until the accuracy reaches 100%. If the addition of a conjunct decreases the accuracy, then the algorithm looks for other conjuncts or stops and starts the iteration of the next rule.

### *Step 4: Next Rule*

After a rule is developed, then all the data points covered by the rule are eliminated from the data set. The above steps are repeated for the next rule to cover the rest of the "+" data points. In Figure 4.23, rule $r_2$ is developed after the data points covered by $r_1$ are eliminated.

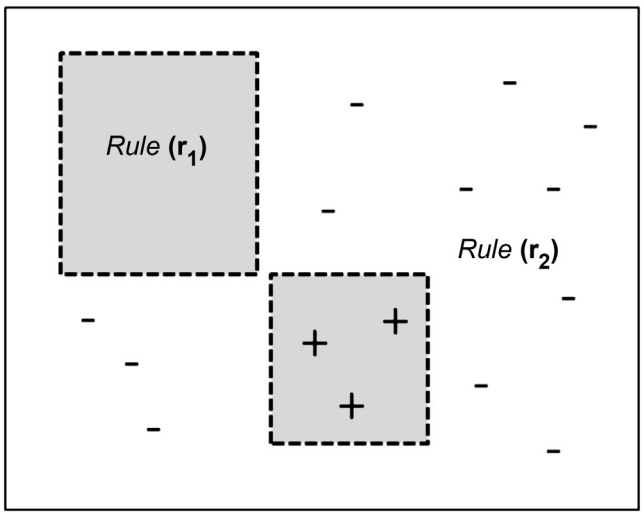



**FIGURE 4.23**
Elimination of $r_1$ data points and next rule.

#### *Step 5: Development of Rule Set*

After the rule set is developed to identify all "+" data points, the rule model is evaluated with a data set used for pruning to reduce generalization errors. The metric used to evaluate the need for pruning is (p – n)/(p + n), where p is the number of positive records covered by the rule and n is the number of negative records covered by the rule. The conjunct is iteratively removed if it improves the metric. All rules to identify "+" data points are aggregated to form a rule group. In multi-class problem, the previous steps are repeated with for next class label. Since this is a two class problem, any data points not covered by the rule set for identifying "+" are predicted to be "–". The outcome of the sequential covering or RIPPER algorithm is a set of optimized rules that can describe the relationship between attributes and the class label, which is also the predictive classification model (Saian & Ku-Mahamud, 2011).

### 4.2.3 How to Implement

Rules remain the most common expression to communicate the inherent relationship in the data. There are a few different ways to generate rules from the data using RapidMiner. The modeling operators for *rule induction* are available in the *Modeling*> *Classification and Regression* > *Rule Induction* folder. The following modeling operators available:

- **Rule Induction:** Commonly used generic rule induction modeler based on the RIPPER algorithm.
- **Single Rule Induction (Attribute):** Uses only one attribute in antecedent, usually the attribute with most predictive power.
- **Single Rule Induction:** Generates only one rule with if/else statement.
- **Tree to Rule:** Indirect method of rule generation that is based on underlying decision tree.

Single rule induction is used for quick discovery of the most dominant rule. Because of its simplicity, single rule modeling operators are used to establish a baseline performance for other classification models. We will review the implementation using the *Rule Induction* and *Tree to Rule* modeling operators in RapidMiner.

#### *Step 1: Data Preparation*

The data set we use in the implementation is the standard Iris data set (See Table 3.1 and Figure 3.1) with four attributes, sepal length, sepal width, petal length, and petal width, and a class label to identify the species of flower viz. *Iris setosa*, *Iris versicolor* and *Iris virginica*. The Iris data set is available in the RapidMiner repository under Sample > Data. Since the original data set refers to the four attributes as a1 to a4, we use the *Rename* operator to change the name of

the attributes (not values) so they can be more descriptive. The *Rename* operator is available in Data Transformation > Name and Role modification. Like a decision tree, rule induction can accept both numeric and polynominal data types. The Iris data set is split into two equal sets for training and testing, using the *Split Data* operator (Data Transformation > Filtering > Sampling). The split ratio used in this implementation is 50%-50% for training and test data.

### Step 2: Modeling Operator and Parameters

The Rule Induction modeling operator accepts the training data and provides the rule set as the model output. The rule set is the text output of if-then rules, along with the accuracy and coverage statistics. The following parameters are available in the model operator and can be configured for desired modeling behavior.

- **Criterion:** Since the algorithm takes the greedy strategy, it needs an evaluation criterion to indicate whether adding a new conjunct helps in a rule. Information gain is commonly used for RIPPER and is similar to information gain for decision trees. Another easy-to-use criterion is accuracy, which was discussed in the sequential covering algorithm.
- **Sample ratio:** This is the ratio of data used for training in the example set. The rest of the data is used for pruning. This sample ratio is different from the training/test split ratio that is used in the data preparation stage.
- **Pureness:** This is the minimal ratio of accuracy desired in the classification rule.
- **Minimal prune benefit:** This is the percentage increase in the prune metric required at the minimum.

The output of the model is connected to the *Apply Model* operator to apply the developed rule base against the test data set. The test data set from the *Split Data* operator is connected to the *Apply Model* operator. The *Performance* operator for classification is then used to create the performance vector from the labeled data set generated from the *Apply Model* operator. The process can be saved and executed after the output ports are connected to the result ports. Figure 4.24 shows the complete RapidMiner process for rule induction. The completed process and the data set can be downloaded from the companion website of the book at www.LearnPredictiveAnalytics.com.

### Step 3: Results Interpretation

The results screen consists of the Rule Model window, the labeled test data set, and the performance vector. The performance vector is similar to the decision tree performance vector. The Rule Model window, shown in Figure 4.25, consists of a sequence of if-then rules along with antecedents and consequents.

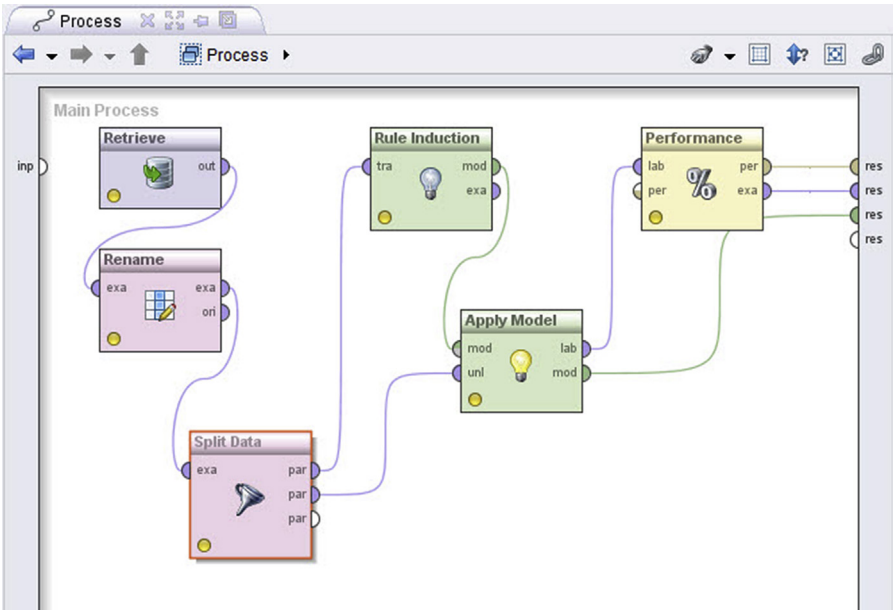



**FIGURE 4.24**
RapidMiner process for rule induction.

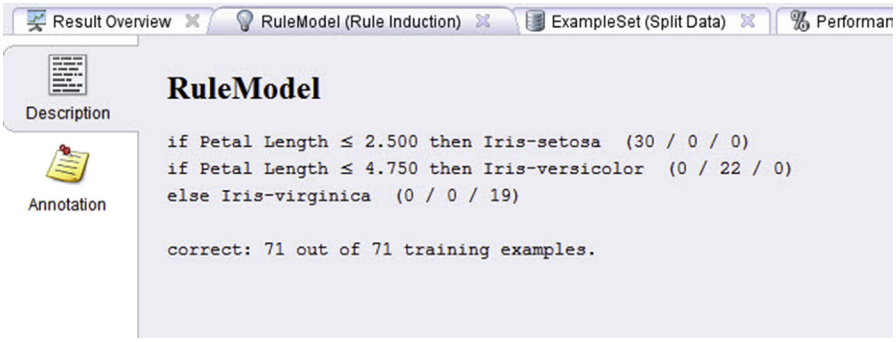



**FIGURE 4.25**
Rule output for rule induction.

The parentheses next to each classification rule indicate the class distribution of the rule covered from the training data set. Note that these statistics are based on the training data set, not the test dataset.

The Performance Vector window provides the accuracy statistics of the prediction based on the rules model for the test data set. For the Iris data set and the RapidMiner process shown in this example, the accuracy of prediction is 92%.

Sixty-nine out of 75 test records are predicted accurately based on simple rules developed by the rule induction model. Not bad for a quick, easy-to-use and easy-to-understand model!

### *Alternative Approach: Tree-to-Rules*

An indirect but easy way to generate a mutually exclusive and exhaustive rule set is to convert a decision tree to an induction rule set. Each classification rule can be traced from the leaf node to the root node, where each node becomes a conjunct and the class label of the leaf becomes the consequent. Even though tree-to-rules may be simple to implement, the resulting rule set may not be the most optimal to understand, as there are many repetitive nodes in the rule path.

In the data mining process developed in RapidMiner, we can just replace the previous Rule Induction operator with the *Tree to Rules* operator. This modeling operator does not have any parameters as it simply converts the tree to rules. However, we have to specify the decision tree in the inner subprocess of the *Tree to Rules* operator. On double-clicking the *Tree to Rules* operator, we can see the inner process where a *Decision Tree* modeling operator has to be inserted as shown in Figures 4.26 and 4.27.

The parameters for a decision tree are the same and reviewed in Section 4.1 of this chapter. The RapidMiner process can be saved and executed. The result set consists of set of rule models, usually with repetitive conjuncts in

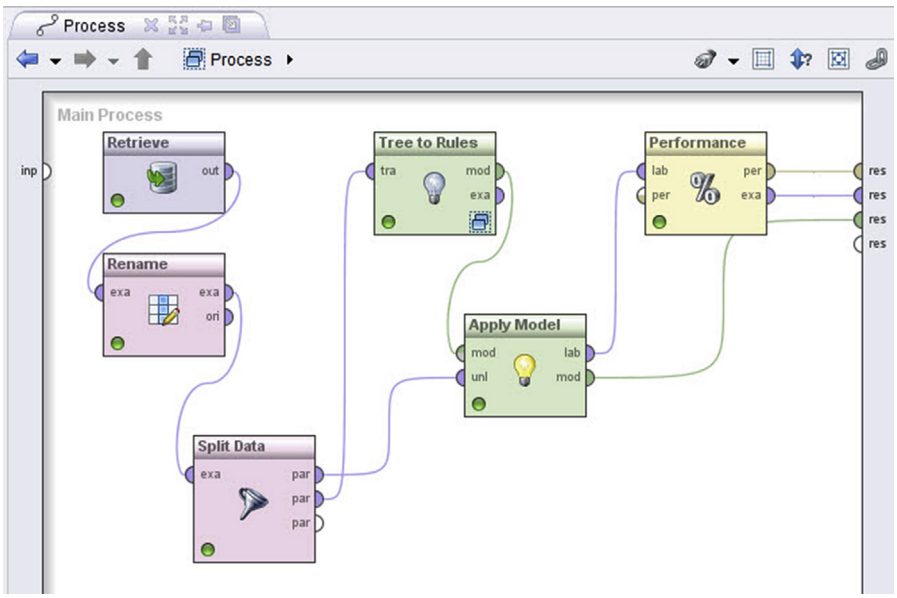



**FIGURE 4.26**
RapidMiner process for Tree to Rules operator.

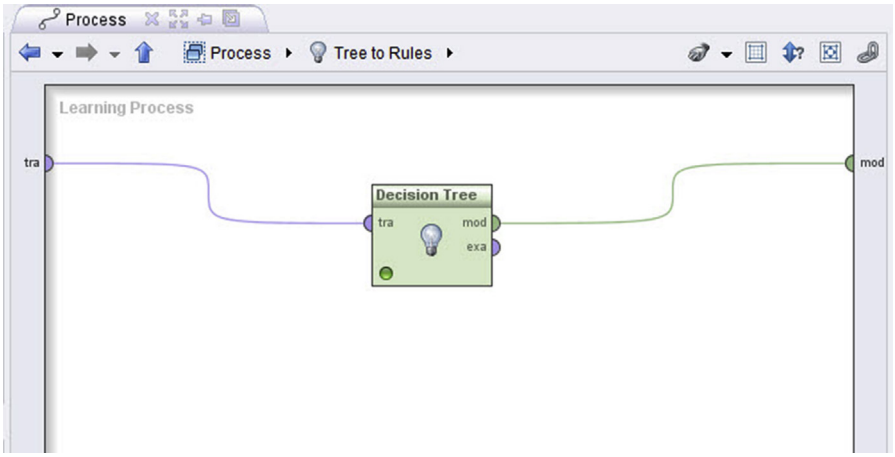



**FIGURE 4.27**
Decision Tree operator inside the subprocess for Tree to Rules.

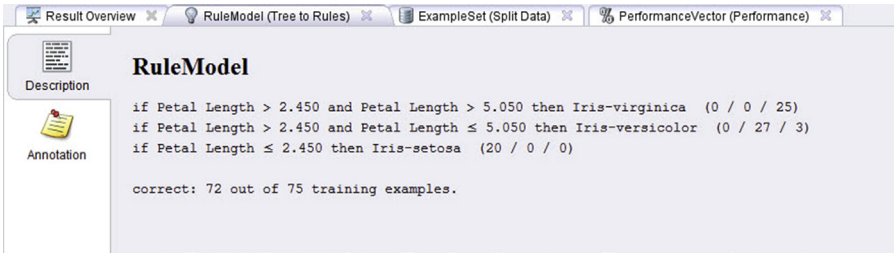



**FIGURE 4.28**
Rules based on decision tree.

antecedents, a fingerprint of rules derived from trees. Note the difference between the rules that are developed for the Rule Induction operator and the rules developed from *Tree to Rules* operator. The rules generated from Tree to Rules are shown in Figure 4.28.

### 4.2.4 Conclusion

Classification using rules provides a simple framework to identify a relationship between factors and the class label that is not only used as a predictive model, but a descriptive model. Rules are closely associated to decision trees. They split the data space in rectilinear fashion and generate mutually exclusive and exhaustive data sets. When the rule set is not mutually exclusive, then the data space can be divided by complex and curved decision boundaries. Single rule learners are the simplest form of data mining model and indicate the most powerful predictor in the given set of attributes. Since rule induction is

a greedy algorithm, the result may not be the most globally optimal solution and like decision trees, rules can overlearn the example set. This scenario can be mitigated by pruning. Given the wide reachability of rules, rule induction is commonly used as a tool to express the results of data mining, even if other data mining algorithms are used.

## 4.3 k-NEAREST NEIGHBORS

The predictive data mining using decision trees and rule induction techniques were built by generalizing the relationship within the data set and using it to predict the outcome of new unseen data. If we need to predict the loan interest rate based on credit score, income level, and loan amount, one approach is to develop a mathematical relationship such as an equation $y = f(X)$ based on the known data and then using the equation to predict interest rate for new data points. These approaches are called *eager learners* because they attempt to find a best approximation of the actual relationship between the input and target variables. But there is also a simple alternative approach. We can "predict" the interest rate for a given credit score, income level and loan amount by *looking up* the interest rate of other customer loan records with similar credit score, closely matching income level and loan amount from the training data set. This alternative class of learners adopts a blunt approach, where no "learning" is performed from the training data set; rather the training data set is used as a *lookup* table to match the input variables and find the outcome. These approaches are called *lazy learners*.

The underlying idea here is somewhat similar to the old adage, "birds of a feather flock together." Similar records congregate in a neighborhood in n-dimensional space, with the same target class labels. This is the central logic behind the approach used by the k-nearest neighbor algorithm, or simply k-NN. The entire training data set is "memorized" and when unlabeled example records need to be classified, the input attributes of the new unlabeled records are compared against the entire training set to find a closest match. The class label of the closest training record is the predicted class label for the unseen test record. This is a nonparametric method, where no generalization or attempt to find the distribution of the data set is made (Altman, 1992). Once the training records are in memory, the classification of the test record is very straightforward. We need to find the closest training record for each test record. Even though no mathematical generalization or rule generation is involved, finding the closet training record for a new unlabeled record can be a tricky problem to solve, particularly when there is no exact match of training data available for a given test data record.

## PREDICTING THE TYPE OF FOREST

Satellite imaging and digital image processing have provided a wealth of data about almost every part of the earth's landscape. There is strong motivation for forestry departments, government agencies, universities, and research bodies to understand the makeup of forests, species of trees and their health, biodiversity, density, and forest condition. Field studies for developing forest databases and classification project are quite a tedious task and expensive. However, we can aid this process by leveraging satellite imagery, limited field data, elevation models, aerial photographs, and survey data (McInerney, 2005). The objective is to classify whether the landscape is a forest or not and further predict the type of trees and species.

The approach to classifying the landscape involves dividing the area into land units (e.g., a pixel in a satellite image) and creating a vector of all the measurements for the land unit, by combining all the available data sets. Each unit's measurements are then compared against the measurements of known pre-classified units. For a given pixel, we can find a pixel in the pre-classified catalog, which has measurements very close to the measurement of the pixel to be predicted. Say the pre-classified pixel with the closest measurement corresponds to birch trees. Thus, we can predict the pixel area to be a birch forest. Every pixel's measurement is compared to measurements of the pre-classified data set to determine the like pixels and hence same forest types. This is the core concept of the k-nearest neighbor algorithm that is used to classify the landscape areas. (Haapanen et al., 2001)

### 4.3.1 How it Works

Any record in a data set can be visualized as a point in an n-dimensional space, where n is the number of attributes. While it is hard for humans to visualize in more than three dimensions, mathematical functions are scalable to any dimension and hence we can perform all the operations that can be done in two-dimensional spaces in the n-dimensional space. Let's consider the standard Iris data set (150 examples, four attributes, one class label. See Figure 3.1 and Table 3.1) and focus on two attributes, petal length and petal width. The scatterplot of these two dimensions is shown in Figure 4.29. The colors indicate the species of Iris, the target class variable. For an unseen test data point A with (petal length, petal width) values (2.1, 0.5), we can deduce visually that the predicted species for the values of data point A would be *Iris setosa*. This is based on the fact that test data point A is in the neighborhood of other data points that belong to species *Iris setosa*. Similarly, unseen test data point B has values (5.7, 1.9) and is in the neighborhood of *Iris versicolor*, hence the test data point can be classified as *Iris versicolor*. However, if the data points are in between the boundaries of two species, for data points such as (5.0, 1.8), then the classification can be tricky because the neighborhood has more than one species in the vicinity. We need an efficient algorithm to resolve these corner cases and measure nearness of data points with more than two dimensions. One technique is to find the nearest training data point from an unseen test data point in multidimensional space, and use the target class value of the nearest training data point as the predicted target class for the test data point. This is similar to how the k-NN algorithm works.

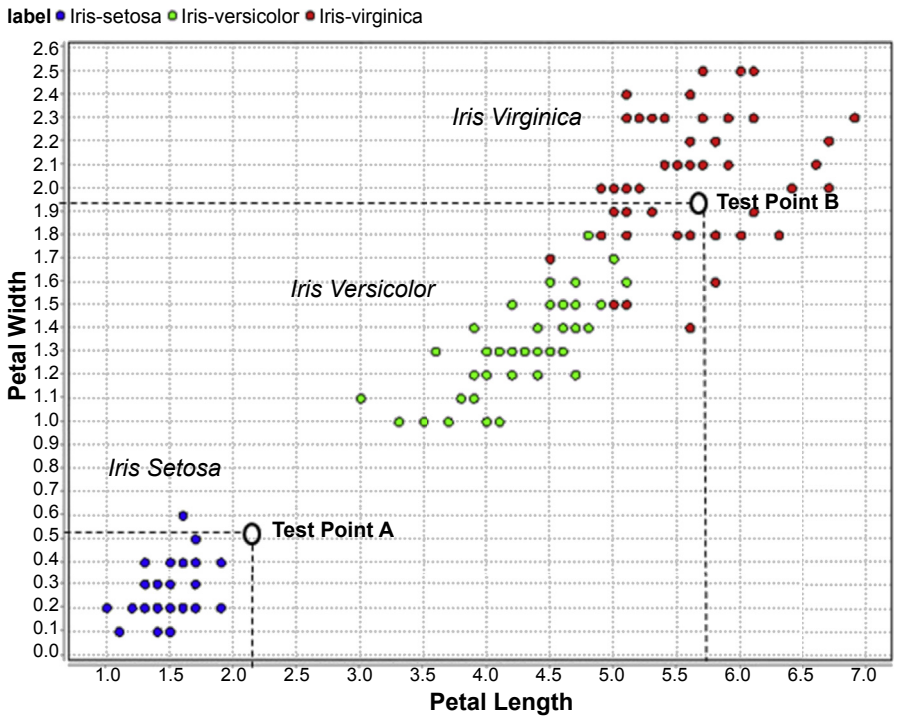



**FIGURE 4.29**

Two-dimensional plot of Iris data set: sepal length and sepal width. Classes are stratified with colors.

The k in the k-NN algorithm indicates the number of close training record(s) that need to be considered when making the prediction for an unlabeled test record. When k = 1, the model tries to find the *first* nearest record and adopts the class label of the first nearest training record as the predicted target class value. Figure 4.30 provides an example training set with two dimensions and the target class values as circles and triangles. The unlabeled test record is the dark square in the center of the scatterplot. With k = 1, the predicted target class value of an unlabeled test record is *triangle* because the closest training record is a triangle. But, what if the closest training record is an outlier with the incorrect class in the training set? Then, all the unlabeled test records near the outlier will get wrongly classified. To prevent this misclassification, we can increase the value of k to, say, 3. When k = 3, we consider the nearest *three* training records instead of one. From Figure 4.30, based on the majority class of the nearest three training records, we can conclude the predicted class of the test record is *circle*. Since the class of the target record is evaluated by voting, k is usually assigned an odd number for a two-class problem (Peterson, 2009).

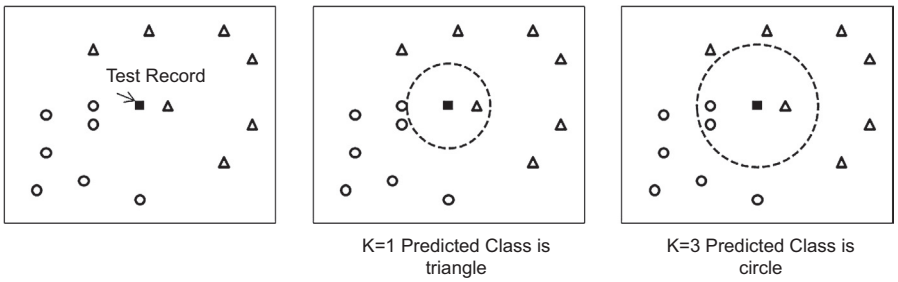



**FIGURE 4.30**

(a) Data set with a record of unknown class. (b) Decision boundary with k = 1 around unknown class record. (c) Decision boundary with k = 3 around unknown test record.

The key task in the k-NN algorithm is determination of the nearest training record from the unlabeled test record using a measure of proximity. Once the nearest training record(s) are determined, the subsequent voting of the nearest training records is straightforward. Let's discuss the various techniques used to measure proximity.

### *Measure of Proximity*

The effectiveness of the k-NN algorithm hinges on the determination of how similar or dissimilar a test record is when compared with the memorized training record. A measure of proximity between two records is measure of proximity of its attributes. To quantify similarity between two records, there is a range of techniques available such as calculating distance, correlation, Jaccard similarity, and cosine similarity (Tan, Michael, & Kumar, 2005).

**A. Distance**

The distance between two points X $(x_1, x_2)$ and Y $(y_1, y_2)$ in two-dimensional space can be calculated by Equation 4.3:

$$\text{Distance } d = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2} \tag{4.3}$$

We can generalize the two-dimensional distance formula shown in Equation 4.3 for n-dimensional space, where X is $(x_1, x_2, \ldots, x_n)$ and Y is $(y_1, y_2, \ldots, y_n)$, as

$$\text{Distance } d = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \bullet\bullet\bullet + (x_n - y_n)^2} \tag{4.4}$$

For example, the first two records of a four-dimensional Iris data set (Fisher, 1936) is X = (4.9, 3.0, 1.4, 0.2) and Y = (4.6, 3.1, 1.5, 0.2). The distance between X and Y is $d = \sqrt{(0.3)^2 + (0.1)^2 + (0.1)^2 + (0)^2} = 0.33$ centimeters. All the attributes in the Iris data set are homogenous in terms of measurements (length of flower parts) and units (centimeters). However

in a typical practical data set, it is common to see attributes in different measures (e.g., credit score, income) and varied units. One problem with the distance approach is that it depends on the scale and units of the attributes. For example, the variance in credit score between two records could be a few hundred points, which is minor in magnitude compared to the variance in income, which could on the order of thousands. Consider two pairs of data points with credit score and annual income in USD. Pair A is (500, $40,000) and (600, $40,000). Pair B is (500, $40,000) and (500, $39,800). The first data point in both the pairs is same. The second data point is different than the first data point, with only one attribute changed. In Pair A, credit score is 600, which is significantly different than 500, while the income is the same. In Pair B, income is down by $200 when compared to the first data point, which is only a 0.5% change. One can rightfully conclude that the data points in Pair B are more similar than the data points in Pair A. However, the distance (Equation 4.4) between data points in Pair A is 100 and the distance between Pair B is 200! The variation in income overpowers the variation in credit score. The same phenomenon can be observed when attributes are measured in different units, scales, etc. To mitigate the problem caused by different measures and units, all the inputs of k-NN are usually normalized, where the data values are rescaled to fit a particular range. Normalizing all the attributes provides a fair comparison between them.

Normalization can be performed by a few different methods. Range transformation rescales values of all the attributes to specified min and max values, usually 0 to 1. *Z*-transformation attempts to rescale all the values by subtracting the mean from each value and dividing the result by the standard deviation, resulting in a transformed set of values that has a mean of 0 and standard deviation of 1. For example, when the Iris data set is normalized using Z-transformation, sepal length, which takes values between 4.3 and 7.9 centimeters, and has a standard deviation of 0.828, is transformed to values between −1.86 and 2.84, with standard deviation 1.

The distance measurement discussed so far is also called *Euclidean distance*, which is the most common distance measure for numeric attributes. In addition to the Euclidean, *Manhattan*, and *Chebyshev* distance measures are sometimes used to calculate the distance between two numeric data points. Let's consider two data points X (1,2) and Y (3,1), as shown in Figure 4.31. The Euclidean distance between X and Y is the straight-line distance between X and Y, which is 2.7. Manhattan distance is the sum of the difference between individual attributes, rather than the root difference square. The Manhattan distance between X and Y is (3 − 1) + (2 − 1) = 3. Manhattan distance is also called the taxi cab distance, due to similarities of the visual path traversed by a vehicle around city blocks (In Figure 4.31, the total distance that is covered by

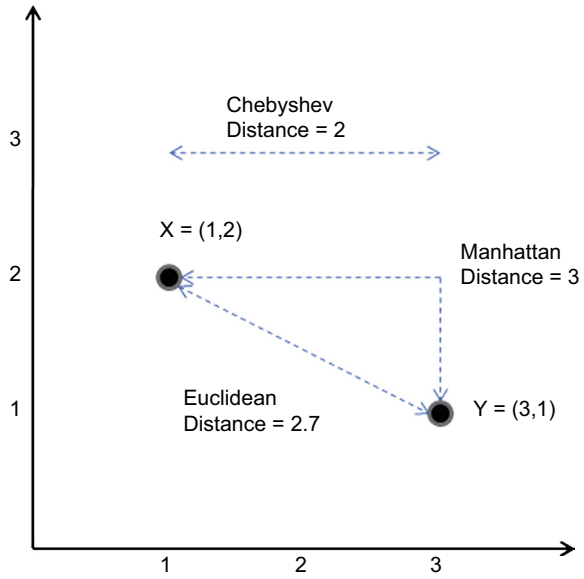



**FIGURE 4.31**
Distance measures.

a cab that has to travel from X to Y in terms of city blocks is two blocks to the right and one block down). Chebyshev distance is the maximum difference between all attributes in the data set. In this example, the Chebyshev distance is the max of [(3 – 1), (1 – 2)] = 2. If Figure 4.31 were a chess board, Chebyshev distance would be the minimum number of moves required by the king to go from one position to another and Manhattan distance is the minimum number of squares covered by the move of a rook from one position to another.

All three aforementioned distance measures can be further generalized by one formula, the ***Minkowski*** distance measure. The distance between two points X ($x_1, x_2, \ldots, x_n$) and Y ($y_1, y_2, \ldots, y_n$) in n-dimensional space is given by Equation 4.5:

$$d = \left( \sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}} \tag{4.5}$$

When $p = 1$, the distance measure is the Manhattan distance, when $p = 2$ the distance measure is the Euclidean distance, and when $p = \infty$ the distance measure is the Chebyshev distance. $p$ is also called the *norm* and Equation 4.5 is called the $p$-norm distance. The choice of distance measure depends on the data (Grabusts, 2011). The Euclidean measure is the most commonly used distance measure for numeric data. Manhattan distance (or Hamming distance) is used for binary attributes. For an unknown (no

prior knowledge) data set, there is no rule-of-thumb distance measure. Euclidean distance would be a good start and the model can be tested with a selection of other distance measures and the corresponding performance. Once the nearest k neighbors are determined, the process of determining the predicted target class is straightforward. The predicted target class is the majority class of the nearest k neighbors. Equation 4.6 provides the prediction of the k-NN algorithm:

$$y' = \text{maximum class}\,(y_1, y_2, \cdots, y_k) \tag{4.6}$$

where $y'$ is the predicted target class of the test data point and $y_i$ is the class of $i^{\text{th}}$ neighbor $n_i$.

**Weights**

The premise of the k-NN algorithm is that data points closer to each other are similar and hence they have the same target class labels. When k is more than one, it can be argued that the closest neighbors should have more say in the outcome of the predicted target class than the farther neighbors (Hill & Lewicki, 2007). The far away neighbors have less influence in determining the final class outcome. This can be accomplished by assigned weights for all the neighbors, with the weights increasing as the neighbors get closer to the data point. The weights are included in the final multivoting step, where the predicted class is calculated. Weights ($w_i$) should satisfy two conditions: they should be proportional to the distance of the data point from the neighbor and the sum of all weights should be equal to one. One of the calculations for weights shown in Equation 4.7 follows an exponential decay based on distance:

$$w_i = \frac{e^{-d(x,n_i)}}{\sum_{i=1}^{k} e^{-d(x,n_i)}} \tag{4.7}$$

where $w_i$ is the weight of $i$th neighbor $n_i$, $k$ the is total number of neighbors, and $x$ is the test data point. The weight is used in predicting target class y′:

$$y' = \text{maximum class}\,(w_1 * y_1, w_2 * y_2, \cdots, w_k * y_k) \tag{4.8}$$

where $y_i$ is the class outcome of neighbor $n_i$.

The distance measure works well for numeric attributes. However, if the attribute is categorical (nominal), the distance between two points is either 0 or 1. If the attribute values are the same, the distance is 0 and if the attribute values are different, the distance is 1. An example distance (overcast, sunny) = 1 and distance (sunny, sunny) = 0. If the attribute is ordinal with more than two values, then the ordinal values can be converted to the integer data type with values 0, 1, 2, …, n – 1 and the converted attribute can be treated as a numeric attribute for distance calculation. Obviously, converting ordinal into numeric retains more information than using it as a categorical data type, where the distance value is either 0 or 1.

**B. Correlation similarity**

Correlation between two data points X and Y is the measure of the linear relationship between the attributes X and Y. Pearson correlation takes the value from −1 (perfect negative correlation) to +1 (perfect positive correlation) with the value of zero being no correlation between X and Y. Since correlation is a measure of "linear" relationship, a zero value doesn't mean there is no relationship. It just means that there is no linear relationship, but there may be a quadratic or any other higher degree relationship between the data points. Also, we are now exploring correlation between one data point and another. *This is quite different from correlation between variables*. Pearson correlation between two data points X and Y is given by

$$\text{Correlation } (\text{X}, \text{Y}) = \frac{s_{xy}}{s_x * s_y} \tag{4.9}$$

where $S_{xy}$ is the covariance of X and Y, which is calculated as $s_{xy} = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})$ and $s_x$ and $s_x$ are the standard deviation of X and Y, respectively. The Pearson correlation of two data points X (1,2,3,4,5) and Y (10,15,35,40,55) is 0.98.

**C. Simple matching coefficient**

The simple matching coefficient (SMC) is used when data sets have binary attributes. For example, let X be (1,1,0,0,1,1,0) and Y be (1,0,0,1,1,0,0). We can measure the similarity between these two data points based on simultaneous occurrence of 0 or 1 with respect to total occurrences. The simple matching coefficient for X and Y can be calculated as follows:

$$\text{Simple matching coefficient } (\text{SMC}) = \frac{\textit{matching occurences}}{\textit{total occurences}}$$

$$= \frac{m_{00} + m_{11}}{m_{10} + m_{01} + m_{11} + m_{00}} \tag{4.10}$$

where

$m_{11}$ = occurrences where X = is 1 and Y = is 1 = 2
$m_{10}$ = occurrences where X = is 1 and Y = is 0 = 2
$m_{01}$ = occurrences where X = is 0 and Y = is 1 = 1
$m_{00}$ = occurrences where X = is 0 and Y = is 0 = 2

$$\text{Simple matching coefficient } (\text{SMC}) = \frac{\textit{matching occurences}}{\textit{total occurences}}$$

$$= \frac{m_{00} + m_{11}}{m_{10} + m_{01} + m_{11} + m_{00}} = \frac{4}{7}$$

**D. Jaccard similarity**

If X and Y represent two text documents, each word can be an attribute in a data set called a term document matrix or document vector. Each

record in the document data set corresponds to a separate document or a text blob. This is explained in greater detail in Chapter 9 Text Mining. In this application, the number of attributes would be very large number, often in the thousands. When comparing two documents X and Y, most of the attribute values will be zero. This means that two documents do not contain the same rare words. In this instance, what is interesting is that the comparison of the *occurrence* of the same word and nonoccurrence of the same word doesn't convey any information and we can ignore it. The Jaccard similarity measure is similar to the simple matching similarity but the nonoccurrence frequency is ignored from the calculation. For the same example X (1,1,0,0,1,1,0) and Y (1,0,0,1,1,0,0),

$$\begin{aligned}\text{Jaccard coefficient} &= \frac{\textit{common occurences}}{\textit{total occurences}} \\ &= \frac{m_{11}}{m_{10}+m_{01}+m_{11}} = \frac{2}{5}\end{aligned} \tag{4.11}$$

**E. Cosine similarity**

We continue the example of the document vectors, where attributes represent either the presence or absence of a word, which takes a binary form of either 1 or 0. It is possible to construct a more informational vector with the number of occurrences in the document, instead of occurrences and nonoccurrences denoted by 1 and 0 respectively. Document data set are usually long vectors with thousands of variables or attributes. For simplicity, consider the example of the vectors with X (1,2,0,0,3,4,0) and Y (5,0,0,6,7,0,0). The cosine similarity measure for two data points is given by

$$\text{Cosine similarity}\,(|\mathrm{X},\mathrm{Y}|) = \frac{x \cdot y}{\|x\|\,\|y\|} \tag{4.12}$$

where $x \cdot y$ is the dot product of the $x$ and $y$ vectors with $X \cdot Y = \sum_{i=1}^{n} x_i y_i$, and $\|\mathrm{x}\|1 = \sqrt{x \cdot x}$. For this example

$$\mathrm{x} \cdot \mathrm{y} = \sqrt{1*5+2*0+0*0+0*6+3*7+4*0+0*0} = 5.1$$
$$\|\mathrm{x}\| = \sqrt{1*1+2*2+0*0+0*0+3*3+4*4+0*0} = 5.5$$
$$\|\mathrm{y}\| = \sqrt{5*5+0*0+0*0+6*6+7*7+0*0+0*0} = 10.5$$
$$\text{Cosine similarity}\,(|\mathrm{x} \cdot \mathrm{y}|) = \frac{x \cdot y}{\|x\|\,\|y\|} = \frac{5.1}{5.5*10.5} = 0.08$$

As with distance measures, there are a few similarity measures available for a categorical attribute in a k-NN implementation. The cosine similarity measure is one of the most used similarity measures, but the determination of the optimal measure comes down to the data structures. The choice of distance or similarity measure can also be parameterized, where multiple models are created with each different measure. The

model with a distance measure that best fits the data with the smallest generalization error can be the appropriate distance measure for the data.

### 4.3.2 How to Implement

Implementation of lazy learners is the most straightforward process amongst all the data mining methods. Since the key functionality here is referencing or looking up the training data set, we could implement the entire algorithm in spreadsheet software like MS Excel, using lookup functions. Of course, if the complexity of the distance calculation or number of attributes rises, then we may need to rely on data mining tools or programming languages. In RapidMiner, k-NN implementation is similar to other classification and regression process, with data preparation, modeling, and performance evaluation operators. The modeling step memorizes all the training records and accepts input in the form of real and nominal values. The output of this modeling step is just the data set of all the training records.

#### *Step 1: Data Preparation*

The data set used in this example is the standard Iris data set with 150 examples and four numeric attributes. First we need to normalize all attributes using the *Normalize* operator, from the Data Transformation > Value Modification > Numerical Value Modification folder. The *Normalize* operator accepts numeric attributes and outputs transformed numeric attributes. The user can specify one of four normalization methods in the parameter configurations: Z-transformation (most commonly used), range transformation, proportion transformation, and interquartile range. In this example, Z-transformation is used because we are standardizing all attributes to same standard deviation.

The data set is then split into two equal exclusive data sets using the *Split Data* operator. Split Data (from Data Transformation > Filtering > Sampling) is used to partition one data set into multiple data sets. The proportion of the partition and the sampling method can be specified in the parameter configuration of the split operator. For this example, the data is split equally between the training and test sets using shuffled sampling. One half of the data set is used as training data for developing the k-NN model and the other half of the data set is used to test the validity of the model.

#### *Step 2: Modeling Operator and Parameters*

The k-NN modeling operator is available in Modeling > Classification > Lazy Modeling. The following parameters can be configured in the operator settings:

- **k:** The value of k in k-NN can be configured. This defaults to one nearest neighbor. This example uses k = 3.
- **Weighted Vote:** In the case of k > 1, this setting determines if the algorithm needs to take into consideration the distance value while predicting the class value of the test record.

- **Measure Types:** There are more than two dozen distance measures available in RapidMiner. These measures are grouped in Measure Types. The selection of Measure Types drives the options for the next parameter (Measure).
- **Measure:** This parameter selects the actual measure like Euclidean distance, Manhattan distance, and so on. The selection of the measure will put restrictions on the type of input the model receives. Depending on the weighting measure, the input data type choices will be limited and hence the data type conversion is required if the input data contains attributes that are not compatible with that measure.

### *Step 3: Evaluation*

Similar to other classification model implementations, we need to apply the model to test the data set, so the effectiveness of the model can be evaluated. Figure 4.32 shows the RapidMiner process where the initial Iris data set is split using a split operator. A random 75 of the initial 150 records are used to build the k-NN model and the rest of the data is the test data set. The *Apply Model* operator takes the test data and applies the k-NN model to predict the class type of the species. The *Performance* operator is then used to compare the predicted class with the labeled class for all of the test records. The complete RapidMiner process can be downloaded from the companion website www.LearnPredictiveAnalytics.com.

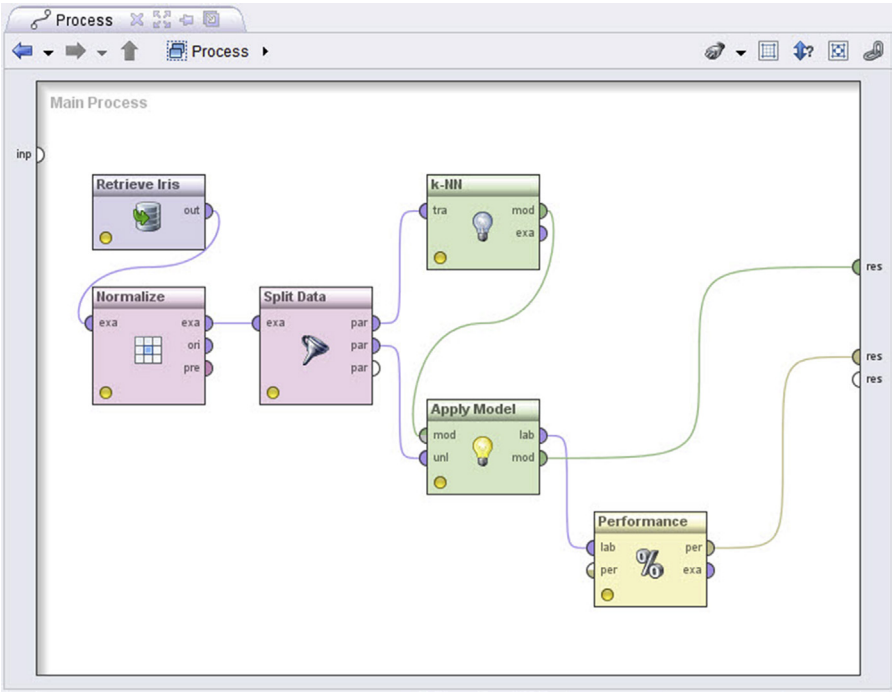



**FIGURE 4.32**
Data mining process for k-NN algorithm.

### *Step 4: Execution and Interpretation*

After the output from the *Performance* operator has been connected to the result ports, as shown in Figure 4.32, the model can be executed. The following result output is observed.

1. **k-NN model:** The model for k-NN is just the set of training records. Hence no additional information is provided in this view, apart from the statistics of training records. Figure 4.33 shows the output model.
2. **Performance vector:** The output of the *Performance* operator provides the confusion matrix with correct and incorrect predictions for all of the test data set. The test set had 75 records. Figure 4.34 shows the accurate prediction of 71 records (sum of diagonal cells in the matrix) and 4 incorrect predictions.

## 4.3.3 Conclusion

The k-NN model requires normalization to avoid any bias by any attribute that has large or small units in the scale. The model is quite robust when there is any missing attribute value in the test record. If the value in the test record is missing, the entire attribute is ignored in the model, and still the model

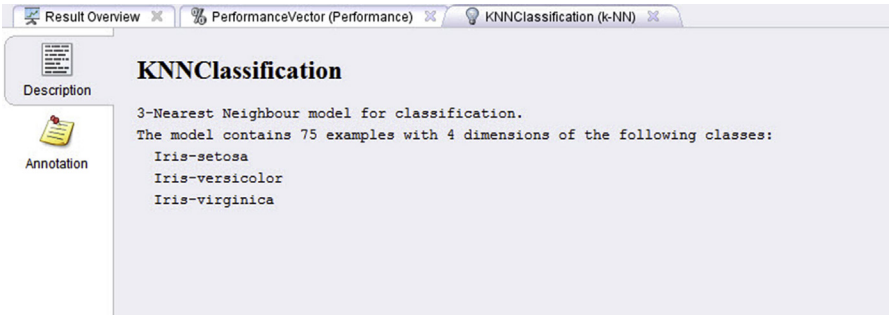
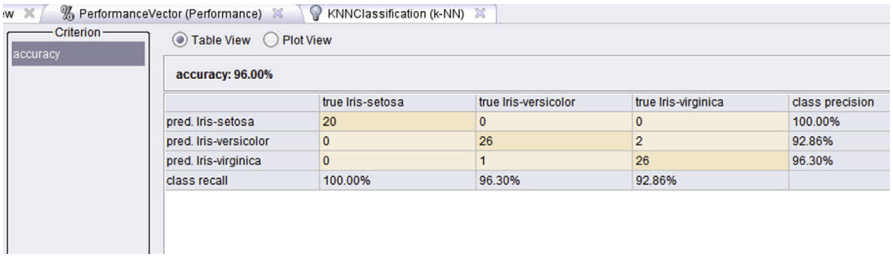



**FIGURE 4.33**
k-NN model output.




| | true Iris-setosa | true Iris-versicolor | true Iris-virginica | class precision |
|---|---|---|---|---|
| pred. Iris-setosa | 20 | 0 | 0 | 100.00% |
| pred. Iris-versicolor | 0 | 26 | 2 | 92.86% |
| pred. Iris-virginica | 0 | 1 | 26 | 96.30% |
| class recall | 100.00% | 96.30% | 92.86% | |

**FIGURE 4.34**
Performance vector for k-NN model.

can function with reasonable accuracy. In the implementation example, if the sepal length of a test record is not known, then the sepal length is ignored in the model. When an attribute value is missing, k-NN becomes a three-dimensional model instead of the regular four dimensions.

As a lazy learner, the relationship between input and output cannot be explained, as the model is just a memorized set of all training records. There is no generalization or abstraction of the relationship. Eager learners are better at explaining the relationship and providing a description of the model.

Model building in k-NN is just memorizing and doesn't require much time. But, when a new unlabeled record is to be classified, the algorithm needs to find the distance between the test record and *all* training records. This process can get very expensive, depending on the size of training set and the number of attributes. A few sophisticated k-NN implementations index the records so that it is easy to search and calculate the distance. We can also convert the actual "real" numbers to ranges so as to make it easy to index and compare it against the test record. However, k-NN is difficult to use in time-sensitive applications like serving an online advertisement or real-time fraud detection.

k-NN models can handle categorical inputs, but the distance between categorical attribute values is either 1, when the attribute values are different or 0, when the attribute values are same. Ordinal values can be converted to integers so that we can better leverage the distance function. Although the k-NN model is not good at generalizing the input-output relationship, it is still quite an effective model for leveraging existing relationships in the training records. For good quality outcomes, it requires a significant number of training records with the maximum possible permutations of input attributes.

## 4.4 NAÏVE BAYESIAN

The data mining algorithms used for classification tasks are quite diverse. The objective of these algorithms is the same—prediction of a target variable. But, the method of predicting is drawn from a range of multidisciplinary techniques. The naïve Bayes algorithm finds its roots in statistics and probability theory. In general, classification techniques try to predict class labels based on attributes by best approximating the relationship between input and output variables. Every day, we mentally estimate a myriad of outcomes based on past evidence. Consider the process of guessing commuting time to work. First, commute time depends heavily on when you are traveling. If you are traveling during peak hours, most likely the commute is going to be longer. Weather conditions like rain, snow, or dense fog will slow down the commute. If the day is a school holiday, like summer break, then the commute will be lighter than on school days. If there is any planned road work ahead, the commute usually takes longer. When more than one adverse factor is at

play, then the commute will be even longer than if it is just one isolated factor. All this if-then knowledge is based on previous experience of commuting when one or more factors come into play. Our experience creates a model in our brain and we mentally run the model before pulling out of the driveway!

Let's take the case of defaults in home mortgages and assume the average overall default rate is 2%. The likelihood of an average person defaulting on their mortgage loan is 2%. However, if a given individual's credit history is above average (or excellent), then the likelihood of their default would be less than average. Furthermore, if we know that the person's annual salary is above average with respect to loan value, then the likelihood of default falls further. As we obtain more evidence on the factors that impact the outcome, we can make improved guesses about the outcome using probability theory. The naïve Bayesian algorithm basically leverages the probabilistic relationship between the factors (attributes) and the class label (outcome). The algorithm makes a strong and sometimes naïve assumption of independence between the attributes, thus its name. The independence assumption between attributes may not always hold true. In some cases, we can assume annual income and credit score are independent of each other. However, in many cases we just don't know. If one of the factors for the default rate is home value, then we have a scenario where both the annual income and home value factors are correlated and thus not independent. Homeowners with high income tend to buy more expensive houses. The independence assumption doesn't hold true always, but the simplicity and robustness of the algorithm offsets the limitation introduced by the independence assumption.

## PREDICTING AND FILTERING SPAM EMAIL

Spam is unsolicited bulk email sent to a wide number of email users. At best it is an annoyance to recipients but many of the spam emails hide a malicious intent by hosting false advertisements or redirecting clicks to phishing sites. Filtering spam email is one of the essential features provided by email service providers and administrators. The key challenge is balance between incorrectly flagging a legitimate email as spam (false positive) versus not catching all the spam messages. There is no perfect spam filtering solution and spam detecting is a catch-up game. The spammers always try to deceive and outsmart the spam filters and email administrators fortify the filters for various new spam scenarios. Automated spam filtering based on algorithms provides a promising solution in containing spam and a learning framework to update the filtering solutions (Prosess Software, 2013).

Some words occur in spam emails more often than in legitimate email messages. For example, the probability of occurrence for words like free, mortgage, credit, sale, Viagra, etc. is higher in spam mails than in normal emails. We can calculate the exact probabilities if we have a sample of previously known spam emails and regular emails. Based on the known word probabilities, we can compute the overall probability of an email being spam based on all the words in the email and the probability of each word being in spam versus regular emails. This is the foundation of Bayesian spam filtering systems (Zdziarski, 2005). Any wrongly classified spam messages that are subsequently reclassified by the user is an opportunity to refine the model, making spam filtering adaptive to new spam techniques. Though recent spam reduction uses a combination of different algorithms, Bayesian-based spam filtering remains one of the foundational elements of spam prediction systems (Sahami et al., 1998).

### 4.4.1 How it Works

The naïve Bayesian algorithm is built on Bayes' theorem, named after Reverend Thomas Bayes. Bayes' work is described in "Essay Towards Solving a Problem in the Doctrine of Chances" (1763), published posthumously in the *Philosophical Transactions of the Royal Society of London* by Richard Price. Bayes' theorem is one of the most influential and important concepts in statistics and probability theory. It provides a mathematical expression for how a degree of subjective belief changes to account for new evidence. First, let's discuss the terminology used in Bayes' theorem.

Assume **X** is the evidence (or factors or attribute set) and Y is the outcome (or target or label class). Here **X** is a set, not individual attributes, hence $\mathbf{X} = \{X_1, X_2, X_3, \ldots, X_n\}$, where $X_i$ is an individual attribute, such as credit rating. The probability of outcome P(Y) is called *prior probability*, which can be calculated from the data set. Prior probability shows the likelihood of an outcome in a given data set. For example, in the mortgage case, P(Y) is the default rate of a home mortgage, which is 2%. P(Y|**X**) is called the *conditional probability*, which provides the probability of an outcome given the evidence when we know the value of **X**. Again, using the mortgage example, P(Y|**X**) is the average rate of default given that an individual's credit history is known. If the credit history is excellent, then the probability of default is likely to be less than 2%. P(Y|**X**) is also called *posterior probability*. Calculating posterior probability is the objective of predictive analytics using Bayes' theorem. This is the likelihood of an outcome as we learn the values of the input attributes.

Bayes' theorem states that

$$P(Y|\mathbf{X}) = \frac{P(Y) * P(\mathbf{X}|Y)}{P(\mathbf{X})} \tag{4.13}$$

P(**X**|Y) is another conditional probability, called the *class conditional probability*. P(**X**|Y) is the probability that an attribute assumes a particular value given the class label. Like P(Y), P(**X**|Y) can be calculated from the data set as well. If we know the training set of loan defaults, we can calculate the probability of an "excellent" credit rating given that the default is a "yes." As indicated in Bayes' theorem, class conditional probability is crucial in calculating posterior probability. P(**X**) is basically the probability of the evidence. In the mortgage example, this is simply the proportion of individuals with a given credit rating. To classify a new record, we can compute P(Y|**X**) for *each class of Y* and see which probability "wins." Class label Y with the highest value of P(Y|**X**) wins for a particular attribute value **X**. Since P(**X**) is the same for every class value of the outcome, we don't have to calculate this and assume it as a constant. More generally, in an example set with n attributes $\mathbf{X} = \{X_1, X_2, X_3 \ldots X_n\}$,

$$P(Y|\mathbf{X}) = \frac{P(Y) * \prod_{i=1}^{n} P(\mathbf{X_i}|Y)}{P(\mathbf{X})} \tag{4.14}$$

If we know how to calculate class conditional probability $P(\mathbf{X}|Y)$ or $\prod_{i=1}^{n} P(X_i \mid Y)$, then it is easy to calculate posterior probability $P(Y|\mathbf{X})$. Since $P(X)$ is constant for every value of Y, it is enough to calculate the numerator of the equation $P(Y) * \prod_{i=1}^{n} P(X_i \mid Y)$ for every class value.

To further explain how the naïve Bayesian algorithm works, let's use the modified Golf data set shown in Table 4.4. The Golf table is an artificial data set with four attributes and one class label. Note that we are using the nominal data table for easier explanation (temperature and humidity have been converted from the numeric type). In Bayesian terms, weather condition is the evidence and decision to play or not play is the belief. Altogether there are 14 examples with 5 examples of Play = no and nine examples of Play = yes. The objective is to predict if the player will Play (yes or no), given the information about a few weather-related measures, based on learning from the data set in Table 4.4. Here is the step-by-step explanation of how the Bayesian model works.

### *Step 1: Calculating Prior Probability P(Y)*

Prior probability P(Y) is the probability of an outcome. In this example set there are two possible outcomes: Play = yes and Play = no. From Table 4.4, out of 14 records there are 5 records with the "no" class and 9 records with the "Yes" class. The probability of outcome is

$P(Y = \text{no}) = 5/14$
$P(Y = \text{yes}) = 9/14$

**Table 4.4** Golf Data Set with Modified Temperature and Humidity Attributes

| No. | Temperature $X_1$ | Humidity $X_2$ | Outlook $X_3$ | Wind $X_4$ | Play (Class Label) Y |
|---|---|---|---|---|---|
| 1 | high | med | sunny | false | no |
| 2 | high | high | sunny | true | no |
| 3 | low | low | rain | true | no |
| 4 | med | high | sunny | false | no |
| 5 | low | med | rain | true | no |
| 6 | high | med | overcast | false | yes |
| 7 | low | high | rain | false | yes |
| 8 | low | med | rain | false | yes |
| 9 | low | low | overcast | true | yes |
| 10 | low | low | sunny | false | yes |
| 11 | med | med | rain | false | yes |
| 12 | med | low | sunny | true | yes |
| 13 | med | high | overcast | true | yes |
| 14 | high | low | overcast | false | yes |

Since the probability of an outcome is calculated from the data set, it is important that the data set used for data mining is *representative* of the population, if sampling is used. A class-stratified sampling of data from the population will not be compatible for naïve Bayesian modeling.

***Step 2: Calculating Class Conditional Probability*** $P(X_i \mid Y)$

Class conditional probability is the probability of *each* attribute value for an attribute, for each outcome value. This calculation is repeated for all the attributes: Temperature ($X_1$), Humidity ($X_2$), Outlook ($X_3$), and Wind($X_4$), and for every distinct outcome value. Let's calculate the class conditional probability of Temperature ($X_1$). For each value of the Temperature attribute, we can calculate $P(X_1|Y = no)$ and $P(X_1|Y = yes)$ by constructing a probability table as shown in Table 4.5. From the data set there are five Y = no records and nine Y = yes records. Out of the five Y = no records, we can also calculate the probability of occurrence when the temperature is high, medium, and low. The values will be 2/5, 1/5, and 2/5, respectively. We can repeat the same process when the outcome Y = yes.

Similarly, we can repeat the calculation to find the class conditional probability for the other three attributes: Humidity ($X_2$), Outlook ($X_3$), and Wind($X_4$). This class conditional probability table is shown in Table 4.6.

**Table 4.5** Class Conditional Probability of Temperature

| **Temperature ($X_1$)** | **$P(X_1|Y = no)$** | **$P(X_1|Y = yes)$** |
|---|---|---|
| high | 2/5 | 2/9 |
| med | 1/5 | 3/9 |
| low | 2/5 | 4/9 |

**Table 4.6** Conditional Probability of Humidity, Outlook, and Wind

| **Humidity ($X_2$)** | **$P(X_1|Y = no)$** | **$P(X_1|Y = yes)$** |
|---|---|---|
| high | 2/5 | 2/9 |
| low | 1/5 | 4/9 |
| med | 2/5 | 3/9 |
| **Outlook ($X_3$)** | **$P(X_1|Y = no)$** | **$P(X_1|Y = yes)$** |
| overcast | 0/5 | 4/9 |
| Rain | 2/5 | 3/9 |
| sunny | 3/5 | 2/9 |
| **Wind ($X_4$)** | **$P(X_1|Y = no)$** | **$P(X_1|Y = yes)$** |
| false | 2/5 | 6/9 |
| true | 3/5 | 3/9 |

**Table 4.7** Test Record

| No. | Temperature $X_1$ | Humidity $X_2$ | Outlook $X_3$ | Wind $X_4$ | Play (Class Label) Y |
|---|---|---|---|---|---|
| Unlabeled Test | high | low | sunny | false | ? |

### *Step 3: Predicting the Outcome Using Bayes' Theorem*

We are all set with preparing class conditional probability tables and now they can be used in the future prediction task. If a new, unlabeled test record (Table 4.7) has the attribute values Temperature= high, Humidity = low, Outlook = sunny, and Wind = false, what would be the class label prediction? Play = Yes or No? The outcome class can be predicted based on Bayes' theorem by calculating the posterior probability P(Y|**X**) for both values of Y. Once P(Y = yes|**X**) and P(Y = no|**X**) are calculated, we can determine which outcome has higher probability and the predicted outcome is the one that has the highest probability. While calculating both class conditional probabilities using Equation 4.14, it is sufficient to just calculate $P(Y) * \prod_{i=1}^{n} P(Xi|Y)$ as P(**X**) is going to be same for both the outcome classes.

$$P(Y = yes|\mathbf{X}) = \frac{P(Y) * \prod_{i=1}^{n} P(Xi|Y)}{P(X)}$$

$$= P(Y = yes) * \{P(Temp = high|Y = yes) * P(Humidity = low|Y = yes) * P(Outlook = sunny|\ Y = yes) * P(Wind = false|Y = yes)\}/P(X)$$

$$= 9/14 * \{2/9 * 4/9 * 2/9 * 6/9\}/P(X)$$

$$= 0.0094/P(X)$$

$$P(Y = no|\mathbf{X}) = 5/14 * \{2/5 * 4/5 * 3/5 * 2/5\}$$

$$= 0.0274/P(X)$$

We normalize both the estimates by dividing both by (0.0094 + 0.027) to get

$$\text{Likelihood of (Play = yes)} = \frac{0.0094}{0.0274 + 0.0094} = 26\%$$

$$\text{Likelihood of (Play = no)} = \frac{0.0094}{0.0274 + 0.0094} = 74\%$$

In this case P(Y = yes|X) < P(Y = no|X), hence the prediction for the unlabeled test record will be Play = no.

Bayesian modeling is relatively simple to understand once you get past the notation (for beginners) and easy to implement in practically any programing language. The computation for model building is quite simple and involves the creation of a lookup table of probabilities. Bayesian modeling is quite robust in handling missing values. If the test example set does not contain a

value, let's suppose temperature is not calculated in the example set, the Bayesian model simply omits the corresponding class conditional probability for the outcomes. Having missing values in the test set would be difficult to handle in decision trees and regression algorithms, particularly when the missing attribute is used higher up in the node of the decision tree or has more weight in regression. Even though the naïve Bayes algorithm is quite robust to missing attributes, it does have a few limitations. Here are couple of the most significant limitations and methods of mitigation.

***Issue 1: Incomplete Training Set***

Problems arise when an attribute value in the testing record has no example in the training record. In the Golf dataset (Table 4.4), if a test example consists of the attribute value Outlook = overcast, the probability of P(Outlook = overcast| Y = no) is zero. Even if one of the attribute's class conditional probabilities is zero, by nature of the Bayesian equation, the entire posterior probability will be zero.

$$\begin{aligned}&P(Y = no|\mathbf{X}) = P(Y = No) * \{P(Temp = high|Y = no) * P(Humidity = \\ &low| Y = no) * P(Outlook = overcast|Y = no) * P(Wind = false|Y = no)\}/P(\mathbf{X}) \\ &= 5/14 * \{2/5 * 1/5 * 0 * 2/5\}/P(\mathbf{X}) \\ &= 0\end{aligned}$$

In this case $P(Y = yes|\mathbf{X}) > P(Y = no|\mathbf{X})$, and the test example will be classified as Play = yes. If there are no training records for any other attribute value, like Temperature = low for outcome *yes*, then probability of both outcomes, $P(Y = no|\mathbf{X})$ and $P(Y = yes|\mathbf{X})$, will also be zero and an arbitrary prediction shall be made because of the dilemma.

To mitigate this problem, we can assign small default probabilities for the missing records instead of zero. With this, the absence of an attribute value doesn't wipe out the value of $P(\mathbf{X}|Y)$, albeit it will reduce the probability to small number. This technique is called Laplace correction. Laplace correction adds a controlled error in all class conditional probabilities.

In the above data set, if the example set contains Outlook = overcast, then P(X|Y = no) = 0. The class conditional probability for all the three values for Outlook is 0/5, 2/5, and 3/5, Y = no. We can add controlled error by adding 1 to all numerators and 3 for all denominators, so the class conditional probabilities are 1/8, 3/8 and 4/8. The sum of all the class conditional probabilities is still 1. Generically, the Laplace correction is given by

$$\text{corrected probability } P(X_i|Y) = \frac{0+\mu p_3}{5+\mu}, \frac{2+\mu p_2}{5+\mu}, \frac{3+\mu p_2}{5+\mu} \tag{4.15}$$

where $p_1 + p_2 + p_3 = 1$ and $\mu$ is the correction.

### *Issue 2: Continuous Attributes*

If an attribute has continuous numeric values instead of nominal values, the solution discussed above will not work. We can always convert the continuous values to nominal values by discretization and use the same approach as discussed. But discretization requires exercising subjective judgment on the bucketing range, leading to loss of information. Instead, we can preserve the continuous values as such and use the probability density function. We assume the probability distribution for a numerical attribute follows a normal or Gaussian distribution. If the attribute value is known to follow some other distribution, such as Poisson, the equivalent probability density function can be used. The probability density function for a normal distribution is given by

$$f(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{\frac{(x-\mu)^2}{2\sigma^2}} \tag{4.16}$$

where $\mu$ is the mean and $\sigma$ is the standard deviation of the sample.

In the updated Golf data set shown in Table 4.8, temperature and humidity are continuous attributes. In such a situation, we compute the mean and standard deviation for both class labels (Play = yes and Play = no) for temperature and humidity (Table 4.9).

If an unlabeled test record has a Humidity value of 78, we can compute the probability density using the Equation 4.16, for both outcomes. For outcome

**Table 4.8** Golf Data Set with Continuous Attributes

| No. | Outlook $X_1$ | Humidity $X_2$ | Temperature $X_3$ | Wind $X_4$ | Play Y |
|---|---|---|---|---|---|
| 1 | sunny | 85 | 85 | false | no |
| 2 | sunny | 80 | 90 | true | no |
| 6 | rain | 65 | 70 | true | no |
| 8 | sunny | 72 | 95 | false | no |
| 14 | rain | 71 | 80 | true | no |
| 3 | overcast | 83 | 78 | false | yes |
| 4 | rain | 70 | 96 | false | yes |
| 5 | rain | 68 | 80 | false | yes |
| 7 | overcast | 64 | 65 | true | yes |
| 9 | sunny | 69 | 70 | false | yes |
| 10 | rain | 75 | 80 | false | yes |
| 11 | sunny | 75 | 70 | true | yes |
| 12 | overcast | 72 | 90 | true | yes |
| 13 | overcast | 81 | 75 | false | yes |

**Table 4.9** Mean and Deviation for Continuous Attributes

| Play Value | | Humidity $X_2$ | Temperature $X_3$ |
|---|---|---|---|
| Y = no | Mean | 74.60 | 84.00 |
| | Deviation | 7.89 | 9.62 |
| Y = yes | Mean | 73.00 | 78.22 |
| | Deviation | 6.16 | 9.88 |

Play = yes, if we plug in the values $x = 78$, $\mu = 73$, and $\sigma = 6.16$ to the probability density function, the equation renders the value 0.04. Similarly for outcome Play = no, we can plug in $x = 78$, $\mu = 74.6$, $\sigma = 7.89$ and compute the probability density to obtain 0.05:

$$P(\text{temperature} = 78|Y = \text{yes}) = 0.04$$
$$P(\text{temperature} = 78|Y = \text{no}) = 0.05$$

The above values are probability densities and *not* probabilities. In a continuous scale, the probability of temperature being exactly at a particular value is zero. Instead, the probability is computed for a range, such as temperatures from 77.5 to 78.5 units. Since the same range is used for computing the probability density for both the outcomes, Play = yes and Play = no, it is not necessary to compute the actual probability. Hence we can substitute the above values in the Bayesian equation 4.14 for calculating class conditional probability $P(\mathbf{X}|Y)$.

### *Issue 3: Attribute Independence*

One of the fundamental assumptions in the naïve Bayesian model is *attribute independence*. Bayes' theorem is guaranteed only for independent attributes. In many real-life cases, this is quite a stringent condition to deal with. This is why the technique is called "naïve" Bayesian, because it assumes attributes independence. However, in practice the naïve Bayesian model works fine with slightly correlated features (Rish, 2001). We can handle this problem by pre-processing the data. Before applying the naïve Bayesian algorithm, it makes sense to remove strongly correlated attributes. In the case of all numeric attributes, this can be achieved by computing a weighted correlation matrix. An advanced application of Bayes' theorem, called a Bayesian belief network, is designed to handle data sets with attribute dependencies.

The independence of two categorical (nominal) attributes can be tested by the *chi-square* ($\chi^2$) test for independence. The chi-square test can be calculated by creating a contingency table of observed frequency like the one shown in Table 4.10A. A contingency table is a simple cross tab of two attributes under consideration.

**Table 4.10** Contingency Tables with **Observed Frequency** (A) and **Expected Frequency** (B)

| | Wind | | | | Wind | | |
|---|---|---|---|---|---|---|---|
| **Outlook** | **False** | **True** | **Total** | **Outlook** | **False** | **True** | **Total** |
| overcast | 2 | 2 | **4** | overcast | 2.29 | 1.71 | **4** |
| rain | 3 | 2 | **5** | rain | 2.86 | 2.14 | **5** |
| sunny | 3 | 2 | **5** | sunny | 2.86 | 2.14 | **5** |
| **Total** | **8** | **6** | **14** | **Total** | **8** | **6** | **14** |

A contingency table of expected frequency (Table 4.10b) can also be created based on the following equation:

$$E_{r,c} = \frac{(\text{row total} * \text{column total})}{(\text{table total})} \tag{4.17}$$

The chi-square statistic ($\chi^2$) calculates the sum of the difference between these two tables. $\chi^2$ is calculated by Equation 4.18. In this equation, O is observed frequency and E is expected frequency:

$$\chi^2 = \sum (O - E)^2/E \tag{4.18}$$

If the chi-square statistic ($\chi^2$) is less than the critical value calculated from the chi-square distribution for a given confidence level, then we can assume the two variables under consideration are independent, for practical purposes. This entire test can be performed in statistical tools or in Microsoft Excel.

### 4.4.2 How to Implement

The naïve Bayesian model is one of the few data mining techniques that can be easily implemented in almost any programing language. Since the conditional probability tables can be prepared in the model building phase, the execution of the model in runtime is very quick. Data mining tools have dedicated naïve Bayes classifier functions. In RapidMiner, *Naïve Bayes* is operator available under *Modeling* > *Classification*. The process of building a model and applying it to new data is similar to decision trees and other classifiers. The *naïve Bayesian* algorithm can accept both numeric and nominal attributes.

#### *Step 1: Data Preparation*

The Golf data set shown in Table 4.8 is available in RapidMiner under Sample > Data in the repository section. The Golf data set can just be clicked and dropped in the process area to source all 14 records of the data set. Within the same repository folder, there is also a Golf-Test data set with a set of 14 records

used for testing. Both data sets need to be added in Main process area. Since the Bayes operator accepts numeric and nominal data types, no other data transformation process is necessary. Sampling is a common method to extract the training data set from a large data set. *It is especially important for naïve Bayesian modeling for the training data set to be representative and proportional to the underlining data set.* Hence, random sampling is recommended instead of class-stratified sampling techniques.

### Step 2: Modeling Operator and Parameters

The *Naïve Bayes* operator (Modeling > Classification) can now be connected to the Golf training data set. The *Naïve Bayesian* operator has only one parameter option to set: whether or not to include Laplace correction. For smaller data sets, Laplace correction is strongly encouraged, as a data set may not have all combinations of attribute values for every class value. In fact, by default, Laplace correction is checked. Outputs of the *Naïve Bayes* operator are the model and original training data set. The model output should be connected to *Apply Model* (Model Application folder) to execute the model on the test data set. The output of the *Apply Model* operator is the labeled test data set and the model.

### Step 3: Evaluation

The labeled data set that we have after using the *Apply Model* operator is then connected to the *Performance – Classification* operator to evaluate the performance of the classification model. The *Performance – Classification* operator can be found under Evaluation > Performance Measurement > Performance. Figure 4.35 shows the complete naïve Bayesian predictive classification process. The output ports can be connected to result ports and the process can be saved and executed. The RapidMiner process is also available for download from the companion website www.LearnPredictiveAnalytics.com.

### Step 4: Execution and Interpretation

The process shown in Figure 4.35 will has three result outputs: a model description, performance vector, and labeled data set. The labeled data set contains the test data set with the predicted class as an added column. The labeled data set also contains the confidence for each label class, which indicates the likelihood of each label class value.

The model description result contains more information on class conditional probabilities of all the input attributes, derived from the training data set. The Charts tab in model description contains probability density functions for the attributes, as shown in Figure 4.36. In the case of continuous attributes, we can discern the decision boundaries across the different class labels for the Humidity attribute. We can see when Humidity exceeds 82, the likelihood of Play =

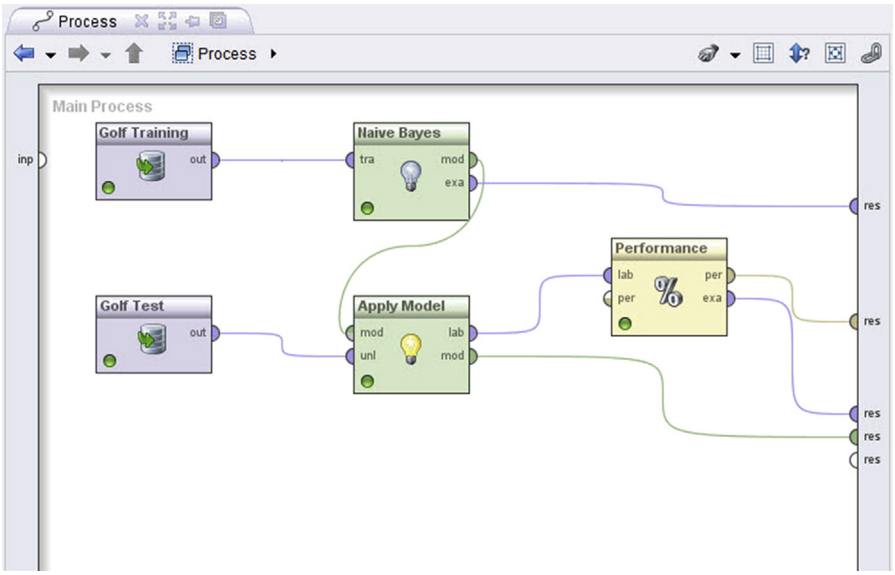



**FIGURE 4.35**
Data mining process for Naive Bayes algorithm.

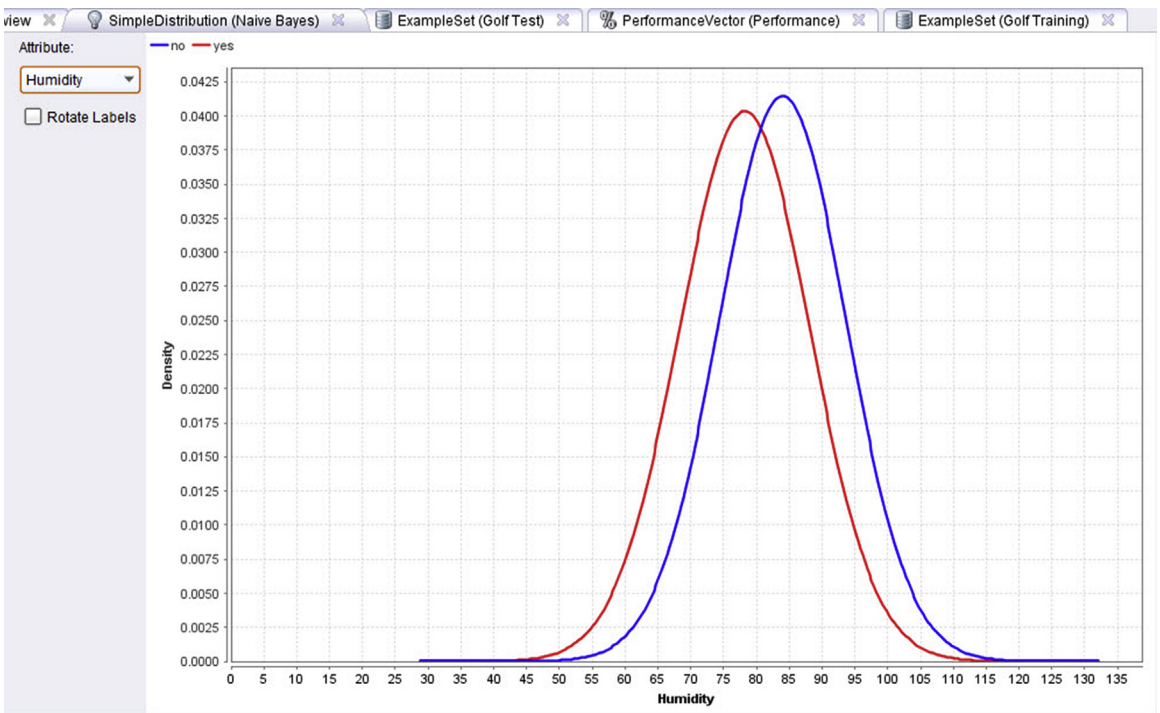
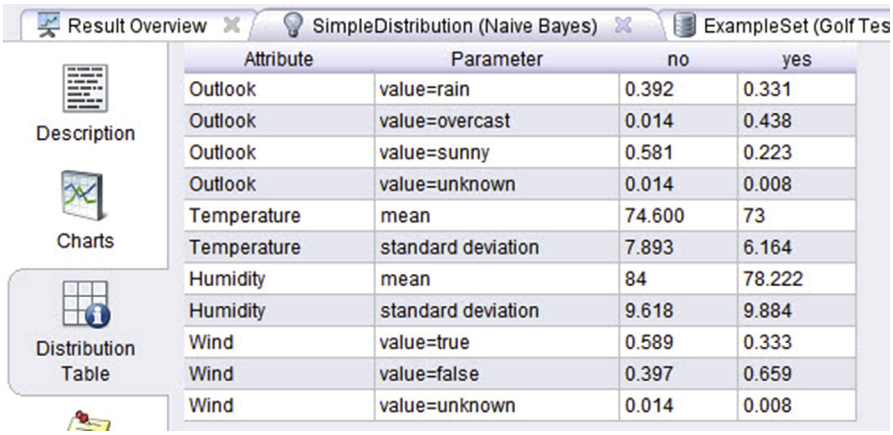



**FIGURE 4.36**
Naive Bayes model output: Probability density function for Humidity attribute.




| Attribute | Parameter | no | yes |
|---|---|---|---|
| Outlook | value=rain | 0.392 | 0.331 |
| Outlook | value=overcast | 0.014 | 0.438 |
| Outlook | value=sunny | 0.581 | 0.223 |
| Outlook | value=unknown | 0.014 | 0.008 |
| Temperature | mean | 74.600 | 73 |
| Temperature | standard deviation | 7.893 | 6.164 |
| Humidity | mean | 84 | 78.222 |
| Humidity | standard deviation | 9.618 | 9.884 |
| Wind | value=true | 0.589 | 0.333 |
| Wind | value=false | 0.397 | 0.659 |
| Wind | value=unknown | 0.014 | 0.008 |

**FIGURE 4.37**
Naive Bayes distribution table output.

no increase. The distribution table is shown in Figure 4.37 with all attribute values and corresponding probability measures. The Distribution Table tab in the model description provides the familiar class conditional probability table similar to Table 4.5 and Table 4.6.

The performance vector output is similar to previously discussed classification algorithms. The performance vector provides the confusion matrix describing accuracy, precision, and recall metrics for the predicted test data set.

### 4.4.3 Conclusion

The Bayesian algorithm provides a probabilistic framework for a classification problem. It has a simple and sound foundation for modeling the data and is quite robust to outliers and missing values. This algorithm is deployed widely in text mining and document classification where the application has a large set of attributes and attribute values to compute. In our experience, the naïve Bayesian classifier is often a great place to start and is very workable as an initial model in the proof of concept (POC) stage for an analytics project. It also serves as a good benchmark for comparison to other models. Implementation of the Bayesian model in production systems is quite straightforward and the use of data mining tools is optional. One major limitation of the model is the assumption of independent attributes, which can be mitigated by advanced modeling or decreasing the dependence across the attributes through preprocessing. The uniqueness of the technique is that it leverages new information as it arrives and tries to make a best prediction considering new evidence. In this way, it is quite similar to how our mind works. Talking about the mind, the next algorithm mimics the biological process of human neurons!

## 4.5 ARTIFICIAL NEURAL NETWORKS

The objective of a predictive analytics algorithm is to model the relationship between input and output variables. The neural network technique approaches this problem by developing a mathematical explanation that closely resembles the biological process of a *neuron*. Although the developers of this technique have used many biological terms to explain the inner workings of neural network modeling process, it has a simple mathematical foundation. Consider the simple linear mathematical model:

$$Y = 1 + 2X_1 + 3X_2 + 4X_3$$

Where Y is the calculated output and $X_1, X_2,$ and $X_3$ are input attributes. 1 is the intercept and 2, 3, and 4 are the scaling factors or coefficients for the input attributes $X_1, X_2,$ and $X_3$, respectively. We can represent this simple linear model in a topological form as shown in Figure 4.38.

In this topology, $X_1$ is the input value and passes through a node, denoted by a circle. Then the value of $X_1$ is multiplied by its weight, which is 2, as noted in the connector. Similarly, all other attributes ($X_2$ and $X_3$) go through a node and scaling transformation. The last node is a special case with no input variable; it just has the intercept. Finally, values from all the connectors are summarized in an output node that yields predicted output Y. The topology shown in Figure 4.38 represents the simple linear model $Y = 1 + 2X_1 + 3X_2 + 4X_3$. The topology also represents a very simple *artificial neural network* (ANN). The neural networks model more complex nonlinear relationships of data and *learn* though adaptive adjustments of weights between the nodes. The ANN is a computational and mathematical model inspired by the biological nervous system. Hence, some of the terms used in an ANN are borrowed from biological counterparts.

In neural network terminology, nodes are called *units*. The first layer of nodes closest to the input is called the input layer or input nodes. The last layer of nodes is called the output layer or output nodes. The output layer performs

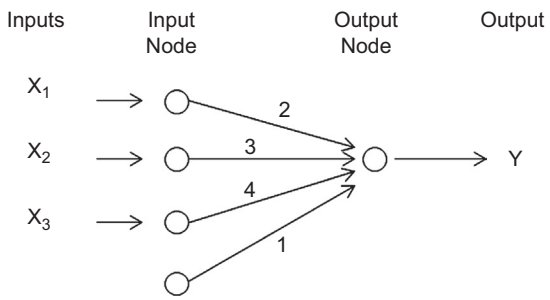



**FIGURE 4.38**
Model topology.

an *aggregation function* and also can have a *transfer function*. The transfer function scales the output into the desired range. Together with the aggregation and transfer function, the output layer performs an *activation function*. This simple two-layer topology, as shown in Figure 4.38, with one input and one output layer is called a *perceptron*, the most simplistic form of artificial neural network. A perceptron is a feed-forward neural network where the input moves in one direction and there are no loops in the topology.

## BIOLOGICAL NEURONS

The functional unit of cells in the nervous system is the neuron. An artificial neural network of nodes and connectors has a close resemblance to a biological network of neurons and connections, with each node acting as a single neuron. There are close to 100 billion neurons in the human brain and they are all interconnected to form this very important organ of the human body (see Figure 4.39). Neuron cells are found in most animals; they transmit information through electrical and chemical signals. The interconnection between one neuron with another neuron happens through a *synapse*. A neuron consists of a cell body, a thin structure that forms from the cell body called dendrite, and a long linear cellular extension called an axon. Neurons are composed of a number of dendrites and one axon. The axon of one neuron is connected to the dendrite of another neuron through a synapse, and electrochemical signals are sent from one neuron to another. There are about 100 trillion synapses in a human brain.

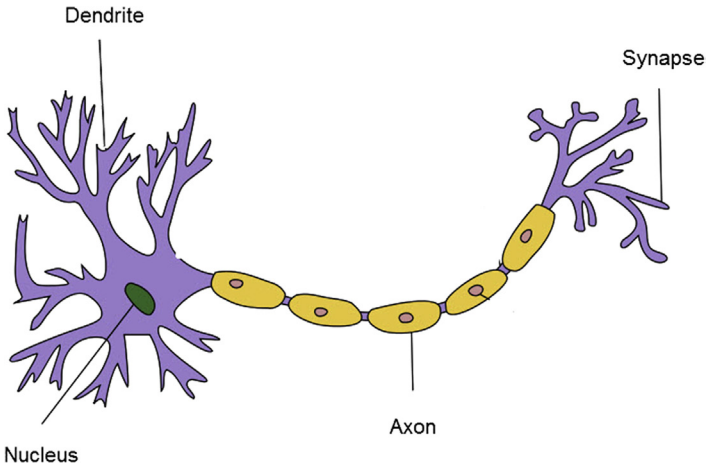



**FIGURE 4.39**

Anatomy of a neuron. (Modified from original "Neuron Hand-tuned." Original uploader: Quasar Jarosz at en.wikipedia.org. Transferred from en.wikipedia.org to Commons by user Faigl.ladislav using CommonsHelper. Licensed under Creative Commons Attribution-Share Alike 3.0 via Wikimedia Commons.[4])

[4]http://commons.wikimedia.org/wiki/File:Neuron_Hand-tuned.svg#mediaviewer/File:Neuron_Hand-tuned.svg

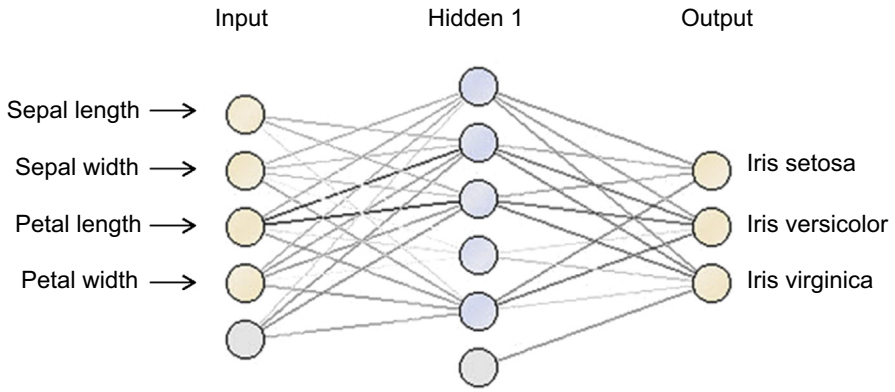



**FIGURE 4.40**
Topology of a neural network model.

A artificial neural network is typically used for modeling *nonlinear*, complicated relationships between input and output variables. This is made possible by the existence of more than one layer in the topology, apart from the input and output layers, called *hidden layers*. A hidden layer contains a layer of nodes that connects input from previous layers and applies an activation function. The output is now calculated by a more complex combination of input values, as shown in Figure 4.40.

Consider the example of the Iris data set. It has four input variables, sepal length, sepal width, petal length, and petal width, with three classes (*Iris setosa*, *Iris versicolor*, *Iris virginica*) in the label. An ANN based on the Iris data set yields a three-layer structure (the number of layers can be specified by the user) with three output nodes, one for each class variable. For a categorical label problem, as in predicting species for Iris, the ANN provides output for each class type. A winner class type is picked based on the maximum value of the output class label. The topology in Figure 4.40 is a feed-forward artificial neural network with one hidden layer. Of course, depending on the problem to be solved, we can use a topology with multiple hidden layers and even with looping where the output of one layer is used as input for preceding layers. Specifying what topology to use is a challenge in neural network modeling and it takes time to build a good approximating model.

The *activation function* used in the output node consists of a combination of an aggregation function, usually summarization, and a *transfer function*. Transfer functions can be anything from sigmoid to normal bell curve, logistic, hyperbolic, or linear functions. The purpose of sigmoid and bell curves is to provide a linear transformation for a particular range of values and a nonlinear transformation for the rest of the values. Because of the

transformation function and the presence of multiple hidden layers, we can model or closely approximate almost any mathematical continuous relationship between input variables and output variables. Hence, a multilayer artificial neural network is called a *universal approximator*. However, the presence of multiple user options such as topology, transfer function, and number of hidden layers makes the search for an optimal solution quite time consuming.

## OPTICAL CHARACTER RECOGNITION

Character recognition is the process of interpreting handwritten text and converting it into digitized characters. It has a multitude of practical applications in our everyday life, including converting handwritten notes to standardized text, automated sorting of postal mail by looking at the zip codes (postal area codes), automated data entry from forms and applications, digitizing classic books, license plate recognition, etc. How does it work?

In its most basic form, character recognition has two steps: digitization and development of the learning model. In the digitization step, every individual character is converted to a digital matrix, say 12x12 pixels, where each cell takes a value of either 0 or 1 based on the handwritten character overlay. The input vector now has 144 binary attributes (12x12) indicating the information of the handwritten characters. Let's assume the objective is to decipher a numeric handwritten zip code (Matan & Kiang, 1990). We can develop an artificial neural network model which accepts 144 inputs and has 10 outputs, each indicating a digit from 0 to 9. The model has to be learned in such a way that when the input matrix is fed, one of the outputs shows the highest signal indicating the prediction for the character. Since a neural network is adaptable and relatively easy to deploy, it is increasingly getting used in character recognition, image processing, and related applications (Li, 1994). This specific use case is also an example where the explanatory aspect of the model is less important—maybe because no one knows exactly how the human brain does it. So there is less expectation that the model should be understandable as long as it works with acceptable performance. This also means ANN models are not easy to explain and in many situations this alone will remove them from consideration of the data mining techniques to use. We wish this wasn't the case!

### 4.5.1 How It Works

An artificial neural network learns the relationship between input attributes and the output class label through a technique called *back propagation*. For a given network topology and activation function, the key training task is to find the weights of the links. The process is rather intuitive and closely resembles the signal transmission in biological neurons. The model uses every training record to estimate the error of the predicted output as compared against the actual output. Then the model uses the error to adjust the weights to minimize the error for the next training record and this step is repeated until the error falls within the acceptable range (Laine, 2003). The rate of correction from one step to other should be managed properly, so that the model does not overcorrect. Following are the key steps in developing an artificial neural network from a training data set.

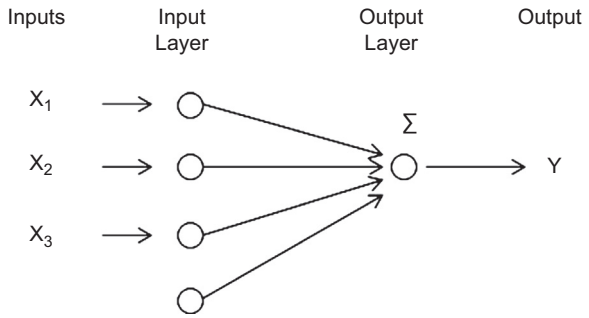



**FIGURE 4.41**
Two-layer topology with summary aggregation.

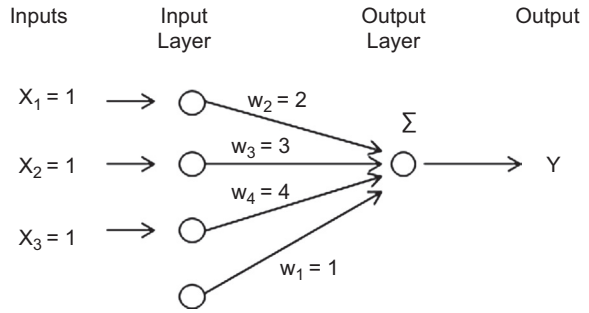



**FIGURE 4.42**
Initiation and first training record.

### *Step 1: Determine the Topology and Activation Function*

For this example, let's assume a data set with three numeric input attributes ($X_1$, $X_2$, $X_3$) and one numeric output (Y). To model the relationship, we are using a topology with two layers and a simple aggregation activation function, as shown in Figure 4.41. There is no transfer function used in this example.

### *Step 2: Initiation*

Let's assume the initial weights for the four links are 1, 2, 3, and 4. Let's take an example model and a test record with all the inputs as 1 and the known output as 15. So, $X_1 = X_2 = X_3 = 1$ and output Y = 15. Figure 4.42 shows initiation of first training record.

### *Step 3: Calculating Error*

We can calculate the predicted output of the record from Figure 4.42. This is simple feed forward process of passing through the input attributes and calculating the predicted output. The predicted output $\overline{Y}$ according to current

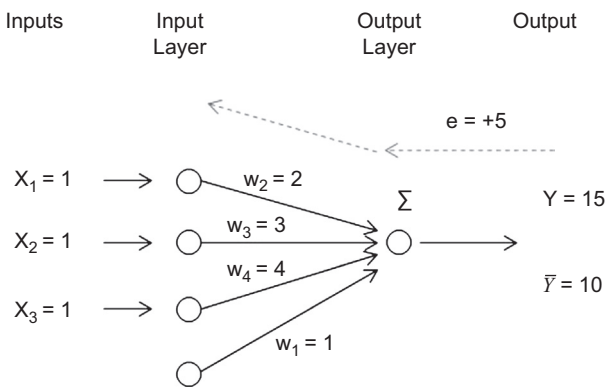



**FIGURE 4.43**
Neural network error back propagation.

model is 1 + 1 * 2 + 1 * 3 + 1 * 4 = 10. The difference between the actual output from training record and predicted output is the error:

$$e = Y - \overline{Y}$$

The error for this example training record is 15 – 10 = 5.

### *Step 4: Weight Adjustment*

Weight adjustment is the most important part of learning in an artificial neural network. The error calculated in the previous step is passed back from the output node to all other nodes in the reverse direction. The weights of the links are adjusted from their old value by a *fraction* of the error. The fraction λ applied to the error is called learning rate. λ takes values from 0 to 1. A value close to 1 results in a drastic change to the model for each training record and a value close to 0 results in smaller changes and less correction. New weight of the link ($w$) is the sum of old weight ($w'$) and the product of learning rate and proportion of the error (λ * $e$).

$$w = w' + \lambda * e$$

The choice of λ can be tricky in the implementation of an ANN. Some model processes start with λ close to 1 and reduce the value of λ while training each cycle. By this approach any outlier records later in the training cycle will not degrade the relevance of the model. Figure 4.43 shows the error propagation in the topology.

The current weight of the first link is $w_2$ = 2. Let's assume the learning rate is 0.5. The new weight will be $w_2$ = 2 + 0.5 * 5/3 = 2.83. The error is divided by 3 because the error is back propagated to three links from the output node. Similarly, the weight will be adjusted for all the links. In the next cycle, a new error

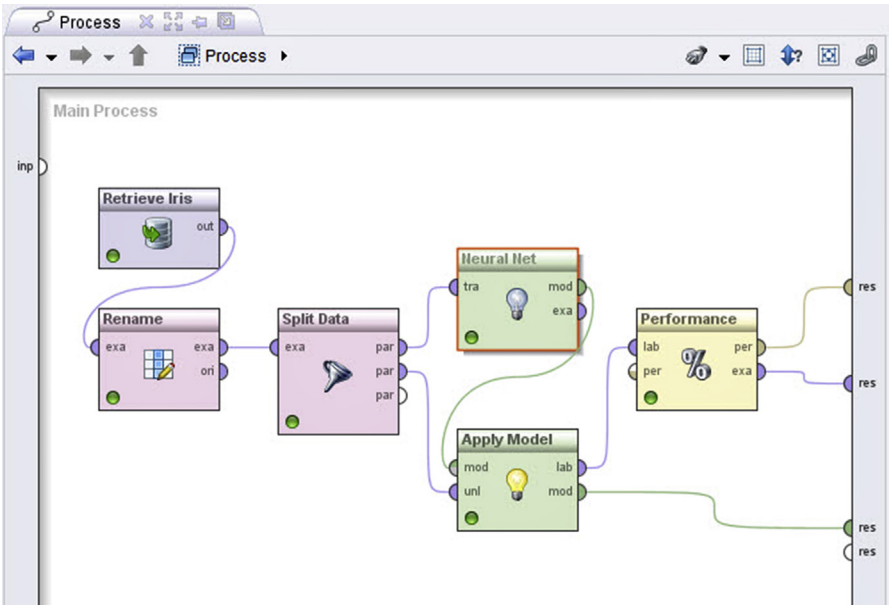



**FIGURE 4.44**
Data mining process for neural network.

will be computed for the next training record. This cycle goes on until all the training records are processed by iterative runs. The same training example can be repeated until the error rate is less than a threshold. We have reviewed a very simple case of an artificial neural network. In reality, there will be multiple hidden layers and multiple output links—one for each nominal class value. Because of the numeric calculation, an ANN model works well with numeric inputs and outputs. If the input contains a nominal attribute, a preprocessing step should be included to convert the nominal attribute into multiple numeric attributes—one for each attribute value, this process is similar to dummy variable introduction, which will be further explored in chapter 10 Time Series Forecasting. This specific preprocessing increases the number of input links for neural network in the case of nominal attributes and thus increases the necessary computing resources. Hence, an ANN is more suitable for attributes with a numeric data type.

## 4.5.2 How to Implement

An artificial neural network is one of the most popular algorithms available for data mining tools. In RapidMiner, the ANN model operators are available in the Classification folder. There are three types of models available: A simple perceptron with one input and one output layer, a flexible ANN model called *Neural Net* with all the parameters for complete model building, and advanced *AutoMLP* algorithm. AutoMLP (for Automatic Multilayer Perceptron) combines

concepts from genetic and stochastic algorithms. It leverages an ensemble group of ANNs with different parameters like hidden layers and learning rates. It also optimizes by replacing the worst performing models with better ones and maintains an optimal solution. For the rest of the discussion, we will focus on the Neural Net model.

### *Step 1: Data Preparation*

The Iris data set is used to demonstrate the implementation of an ANN. All four attributes for the Iris data set are numeric and the output has three classes. Hence the ANN model will have four input nodes and three output nodes. The ANN model will not work with categorical or nominal data types. If the input has nominal attributes, it should be converted to numeric using data transformation, see Chapter 13 Getting Started with RapidMiner. Nominal to binominal conversion operator can be used to convert each value of a nominal attribute to separate binominal attributes. In this example, we use the *Rename* operator to name the four attributes of the Iris data set and the *Split Data* operator to split 150 Iris records equally into the training and test data.

### *Step 2: Modeling Operator and Parameters*

The training data set is connected to the *Neural Net* operator (Modeling > Classification and Regression > Neural Net Training). The *Neural Net* operator accepts real values and later converts them into the normalized range –1 to 1 and outputs a standard ANN model. The following parameters are available in ANN for users to change and customize in the model.

- **Hidden layer:** Determines the number of layers, size of each hidden layer, and names of each layer for easy identification in the output screen. The default size of the node is –1, which is actually calculated by (*number of attributes* + *number of classes*)/2 + 1. The default can be overwritten by specifying an integer of nodes, not including a no-input threshold node per layer.
- **Training cycles:** This number of times a training cycle is repeated; it defaults to 500. In a neural network, every time a training record is considered, the previous weights are quite different and hence it is necessary to repeat the cycle many times.
- **Learning rate:** The value of λ determines how sensitive the change in weight has to be in considering error for the previous cycle. It takes a value from 0 to 1. A value closer to 0 means the new weight would be more based on the previous weight and less on error correction. A value closer to 1 would be mainly based on error correction.
- **Momentum:** This value is used to prevent local maxima and seeks to obtain globally optimized results by adding a fraction of the previous weight to the current weight.

- **Decay:** During the neural network training, ideally the error would be minimal in the later portion of the record sequence. We don't want a large error due to any outlier records in the last few records, thereby impacting the performance of the model. Decay reduces the value of the learning rate and brings it closer to zero for the last training record.
- **Shuffle:** If the training record is sorted, we can randomize the sequence by shuffling it. The sequence has an impact in the model, particularly if the group of records exhibiting nonlinear characteristics are all clustered together in the last segment of the training set.
- **Normalize:** Nodes using a sigmoid transfer function expect input in the range of –1 to 1. Any real value of the input should be normalized in an ANN model.
- **Error epsilon:** The objective of the ANN model should be to minimize the error but not make it zero, at which the model memorizes the training set and degrades the performance. We can stop the model building process when the error is less than a threshold called the error epsilon.

The output of the *Neural Net* operator can be connected to the *Apply Model* operator, which is standard in every predictive analytics workflow. The *Apply Model* operator also gets an input data set from the *Split data* operator for the test data set. The output of the *Apply Model* operator is the labeled test data set and the ANN model.

### *Step 3: Evaluation*

The labeled data set output after using the *Apply Model* operator is then connected to the *Performance – Classification* operator (Evaluation > Performance Measurement > Performance), to evaluate the performance of the classification model. Figure 4.44 shows the complete artificial neural network predictive classification process. The output connections can be connected to the result ports and the process can be saved and executed.

### *Step 4: Execution and Interpretation*

The output results window for the model provides a visual on the topology of the ANN model. Figure 4.45 shows the model output topology. Upon a click on a node, we can get the weights of the incoming links to the node. The color of the link indicates relative weights. The description tab of the model window provides the actual values of the link weights.

The output performance vector can be examined to see the accuracy of the artificial neural network model built for the Iris data set. Figure 4.46 shows the performance vector for the model. A three-layer ANN model with the default parameter options and equal splitting of input data and training set yields 93% accuracy. Out of 75 examples, only 5 were misclassified.

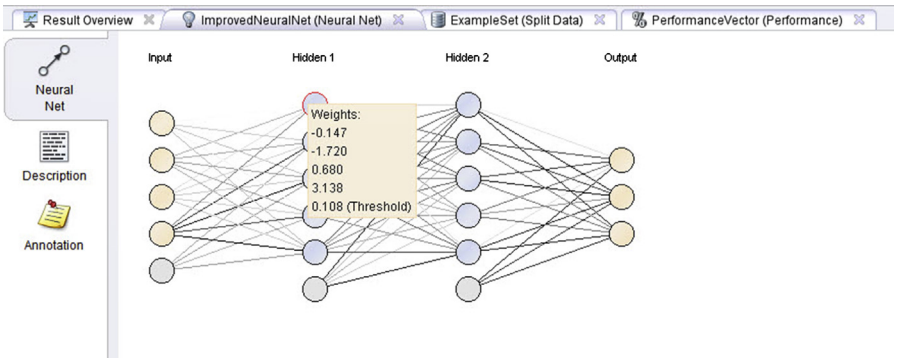
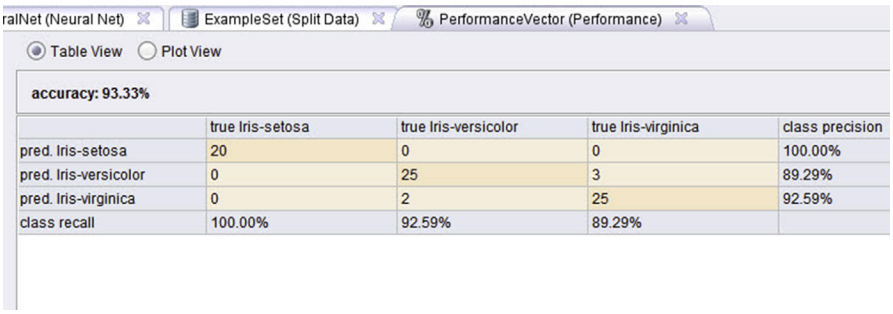



**FIGURE 4.45**
Neural network model output with three hidden layers and four attributes.




| | true Iris-setosa | true Iris-versicolor | true Iris-virginica | class precision |
|---|---|---|---|---|
| pred. Iris-setosa | 20 | 0 | 0 | 100.00% |
| pred. Iris-versicolor | 0 | 25 | 3 | 89.29% |
| pred. Iris-virginica | 0 | 2 | 25 | 92.59% |
| class recall | 100.00% | 92.59% | 89.29% | |

**FIGURE 4.46**
Performance vector for artificial neural network.

### 4.5.3 Conclusion

Neural network models require strict preprocessing. If the test example has missing attribute values, the model cannot function, similar to regression or decision trees. The missing values can be replaced with average values or any default values to minimize the error. The relationship between input and output cannot be explained clearly by an artificial neutral network. Since there are hidden layers, it is quite complex to understand the model. In many data mining applications explanation of the model is as important as prediction itself. Decision trees, induction rules, and regression do a far better job at explaining the model.

Building a good ANN model with optimized parameters takes time. It depends on the number of training records and iterations. There are no consistent guidelines on the number of hidden layers and nodes within each hidden layer. Hence, we would need to try out many parameters to optimize the selection of parameters. However, once a model is built, it is straightforward to implement and an example record gets classified quite fast.

An ANN does not handle categorical input data. If the data has nominal values, it needs to be converted to binary or real values. This means one input attribute explodes to multiple input attributes and exponentially increases nodes, links, and complexity. Also, converting nonordinal categorical data, like zip code, to numeric provides an opportunity for ANN to make numeric calculations, which doesn't quite make sense. Having redundant correlated attributes is not going to be a problem in an ANN model. If the example set is large, having outliers will not degrade the performance of the model. However, outliers will impact the normalization of the signal, which most ANN models require for input attributes. Because model building is by incremental error correction, ANN can yield local optima as the final model. This risk can be mitigated by managing a momentum parameter to weigh the update.

Although model explanation is quite difficult with an ANN, the rapid classification of test examples makes an ANN quite useful for anomaly detection and classification problems. An ANN is commonly used in fraud detection, a scoring situation where the relationship between inputs and output is nonlinear. If there is a need for something that handles a highly nonlinear landscape along with fast real-time performance, then the artificial neural network is a good option.

## 4.6 SUPPORT VECTOR MACHINES

Support vector algorithms are a relatively recent concept, like so many other machine learning techniques. Cortes and Vapnik (Cortes, 1995) provided one of the first formal introductions to the concept while investigating algorithms for optical character recognition at the AT&T Bell Labs.

The term "support vector machine" is a confusing name for a predictive analytics *algorithm*. The fact is this term is very much a misnomer: there is really no specialized hardware. But it is a powerful algorithm that has been very successful in applications ranging from pattern recognition to text mining. A support vector machine (SVM) emphasizes the interdisciplinary nature of today's advanced analytics by drawing equally from three major areas: computer science, statistics, and mathematical optimization theory.

We start with essential terminology and definitions that are unique to SVMs. We will then conceptually explain the functioning of the algorithm for a simple "linear" data set and then a slightly more complex nonlinear dataset. We will provide a brief mathematical explanation of the workings of the algorithm before illustrating how to implement SVMs in practice with a case study. Finally we will highlight how SVMs perform better in some situations compared to other classification techniques and close out this section with a list of the advantages and disadvantages of SVMs in general.

### 4.6.1 Concept and Terminology

At a very basic level, a *support vector machine* is a classification method. It works on the principle of fitting a boundary to a region of points that are all alike (that is, belong to one class). Once a boundary is fitted (on the training sample), for any new points (test sample) that need to be classified, we must simply check whether they lie inside the boundary or not. The advantage of an SVM is that once a boundary is established, most of the training data is redundant. All it needs is a *core set of points* that can help identify and fix the boundary. These data points are called *support vectors* because they "support" the boundary. Why are they called vectors? Clearly because each data point (or observation) is a vector: that is, it is a row of data that contains values for a number of different attributes.

This boundary is traditionally called a *hyperplane*. In a simple example of two dimensions (two attributes), this boundary can be a straight line or a curve (as shown in Figure 4.47). In three dimensions it can be a plane or an irregular complex surface. Higher dimensions are obviously impossible to visualize and a hyperplane is thus a generic name for a boundary in more than three dimensions.

As seen in Figure 4.47, a number of such hyperplanes can be found for the same data set. Which one is the "correct" one? Clearly a boundary that separates the classes with minimal misclassification is the best one. In the above sequence of images shown in Figure 4.47, the algorithm applied to the third image appears to have zero misclassification and may be the best one. Additionally, a boundary line that ensures that the average geometric distance between the two regions (or classes) is maximized is even better. This (n-dimensional) distance is called a *margin*. An SVM algorithm therefore essentially runs an optimization scheme to maximize this margin. The points with the "X" through them are the support vectors.

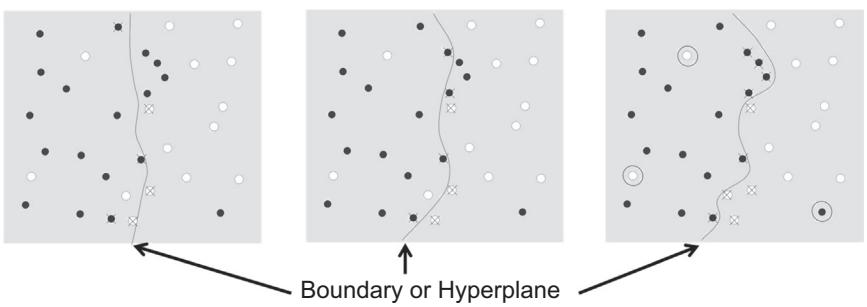



**FIGURE 4.47**

Three different hyperplanes for the same set of training data. There are two classes in this data set, which are shown as filled and open circles.

But it is not always possible to ensure that data can be cleanly separated. It may be rare to find that the data are *linearly separable*. When this happens, there may be many points within the margin. In this case the best hyperplane is the one that has a *minimum* number of such points within the margin. To ensure this, a *penalty* is charged for every "contaminant" inside the margin and the hyperplane that has a minimum aggregate penalty cost is chosen. In Figure 4.48, $\xi$ represents the penalty that is applied for each error and the sum of all such errors is minimized to get the best separation.

What would happen if the data are not linearly separable (even without such contaminating errors)? For example, in Figure 4.49a, the data points belong to two main classes: an inner ring and an outer ring. We know that these two classes are not "linearly separable." In other words we cannot draw a straight line to split the two classes. However, it is intuitively clear that an elliptical or circular "hyperplane" can easily separate the two classes. In fact, if we were to run a simple "linear" SVM on this data, we would get a classification accuracy of around 46%.

How can we classify such complex feature spaces? In the above example, a simple trick would be to transform the two variables $x$ and $y$ into a new feature space involving $x$ (or $y$) and a new variable $z$ defined as $z = \sqrt{(x^2 + y^2)}$. The representation of $z$ is nothing more than the equation for a circle. When the data is transformed in this way, the resulting feature space involving $x$ and $z$ will appear as shown in Figure 4.49b. The two clusters of data correspond to the two radii of the rings: the inner one with an average radius of around 5.5 and the outer cluster with an average radius of around 8.0. This problem is explored in greater detail in a case study later on in this chapter.

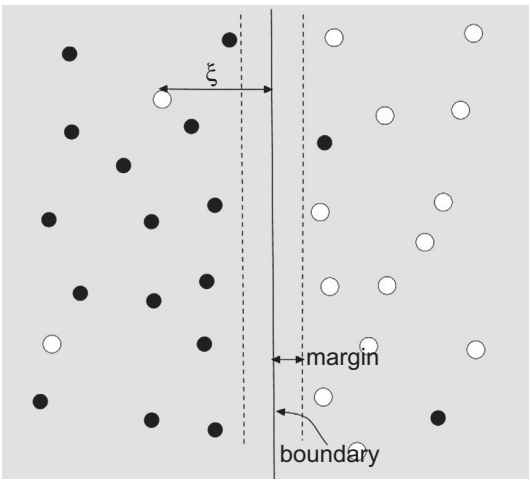



**FIGURE 4.48**
Key concepts in SVM construction: boundary, margin, and penalty, $\xi$.

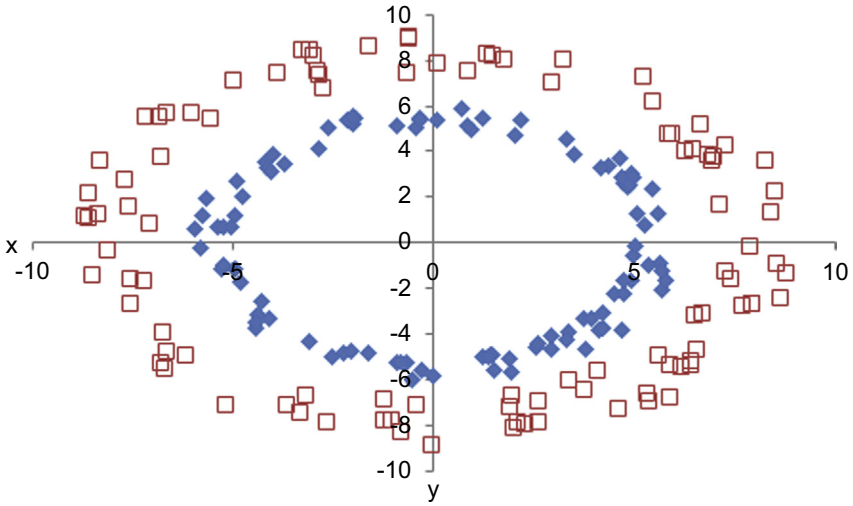



**FIGURE 4.49a**
Linearly nonseparable classes.

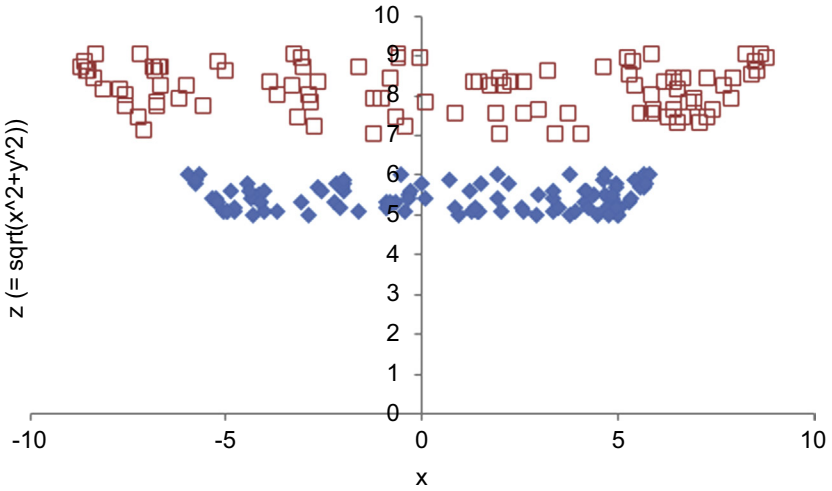



**FIGURE 4.49b**
Transformation to linearly separable.

Clearly this new problem in the $x$ and $z$ dimensions is now linearly separable and we can apply a standard SVM to do the classification. When we run a linear SVM on this transformed data, we get a classification accuracy of 100%. After classifying the transformed feature space, we can invert the transformation to get back our original feature space.

*Kernel functions* offer the user the option of transforming nonlinear spaces into linear ones. Most packages that include SVM will have several nonlinear kernels ranging from a simple polynomial basis functions to sigmoid functions. The user does not have to do the transformation beforehand, but simply has to select the appropriate kernel function; the software will take care of transforming the data, classifying it, and retransforming the results back into the original space.

Unfortunately with a large number of attributes in a data set it is difficult to know which kernel would work best. The most commonly used ones are polynomial and radial basis functions. From a practical standpoint, it is a good idea to start with a quadratic polynomial and work your way up into some of the more exotic kernel functions until we reach a desired accuracy level. This flexibility of support vector machines does come at the price of cost of computation.

Now that we have an intuitive understanding of how SVMs work, we can examine the working of the algorithm with a more formal mathematical explanation.

### 4.6.2 How it Works

Given a training data set, how do we go about determining the boundary and the hyperplane? Let us use the case of a simple linearly separable dataset consisting of two attributes, $\mathbf{x}_1$ and $\mathbf{x}_2$. We know that ultimately by using proper kernels any complex feature space can be mapped into a linear space, so this formulation will apply to any general data set. Furthermore, extending the algorithm to more than two attributes is conceptually straightforward.

There are three essential tasks involved here: the first step is to find the boundary of each class. Then the best hyperplane, **H**, is the one that maximizes the margin or the distance to each of the class boundaries (see Figure 4.48). Both of these steps use the training data. The final step is to determine on which side of this hyperplane a given test example lies in order to classify it.

**Step 1:** Finding the boundary. When we connect every point in one class of a data set to every other in that class, the outline that emerges defines the boundary of this class. This boundary is also known as the *convex hull*, as shown in Figure 4.50.

Each class will have its own convex hull and because the classes are (assumed to be) linearly separable, these hulls do not intersect each other.

**Step 2:** There are infinitely many available hyperplanes, two of which are shown in Figure 4.51. How do we know which hyperplane maximizes the margin? Intuitively we know that $\mathbf{H}_0$ has a larger margin than $\mathbf{H}_1$, but how can this be determined mathematically?

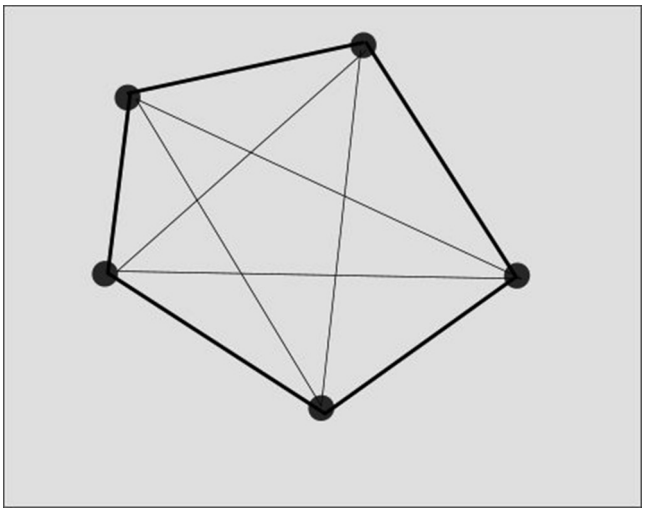


**FIGURE 4.50**
A convex hull for one class of data.

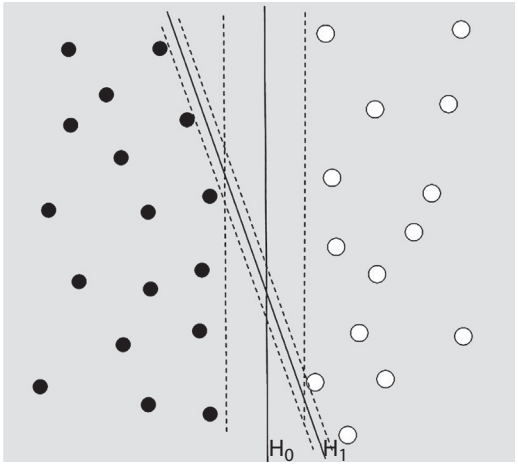



**FIGURE 4.51**
Both hyperplanes shown can separate data. It is intuitively clear that $\mathbf{H_0}$ is better.

First of all, any hyperplane can be expressed in terms of the two attributes, $x_1$ and $x_2$, as follows:

$$\mathbf{H} = b + \mathbf{w} \cdot \mathbf{x} = 0 \tag{4.19}$$

where $\mathbf{x}$ is $(x_1, x_2)$, the weight $\mathbf{w}$ is $(w_1, w_2)$, and $b_0$ is an intercept-like term usually called the "bias." Note that this is similar to the standard form of

the equation of a line. An optimal hyperplane, $\mathbf{H_0}$, is uniquely defined by $(b_0 + \mathbf{w_0} \bullet \mathbf{x} = \mathbf{0})$. Once we define the hyperplane in this fashion, it can be shown that the margin is given by (Cortes, 1995)

$$\text{Margin} = 2/(\sqrt{w_0} \cdot w_0) \tag{4.20}$$

Maximizing this quantity requires quadratic programming, which is a well-established process in mathematical optimization theory (Fletcher, 1987). Furthermore, the $\mathbf{w}_0$ can be conveniently expressed in terms of only a few of the training examples, known as support vectors, as follows:

$$|\mathbf{w}_0 = \sum y_{ii}\mathbf{x}_i| \tag{4.21}$$

where the $y_i$ are the class labels (+1 or −1 for a binary classification), and the $\mathbf{x}_i$ are called the support vectors. The i's are coefficients that are nonzero only for these support vectors.

**Step 3:** Once we have defined the boundary and the hyperplane, any new test example can be classified by computing on which side of the hyperplane the example lies. This is easily found by substituting the test example, **x**, into the equation for the hyperplane. If it computes to +1, then it belongs to the positive class and if it computes to −1 it belongs to the negative class. The interested reader is referred to Smola (2004) or Cortes (1995) for a full mathematical description of the formulation. Hsu (2003) provides a more practical demonstration of programming an SVM.

### 4.6.3 How to Implement

We will now show how RapidMiner determines the classes for two simple cases. *(Note: We have deliberately chosen a pair of simplistic datasets to illustrate how SVMs can be implemented. A more sophisticated case study will be used to demonstrate how to use SVMs for text mining in Chapter 9.)*

#### *Example 1: Applying an SVM to a Simple Linearly Separable Example*

The default SVM implementation in RapidMiner is based on the so-called "dot product" formulation shown in equations above. In this first example we will build an SVM using a two-dimensional data set that consists of two classes: A and B (Figure 4.52). A RapidMiner process reads in the training data set, applies the default SVM model, and then classifies new points based on the model trained.

The dataset consists of 17 rows of data for three attributes: $\mathbf{x_1}$, $\mathbf{x_2}$ and **class**. The attributes $\mathbf{x_1}$ and $\mathbf{x_2}$ are numeric and **class** is a binomial variable consisting of the two classes A and B. Table 4.11 shows the full data set and Figure 4.52

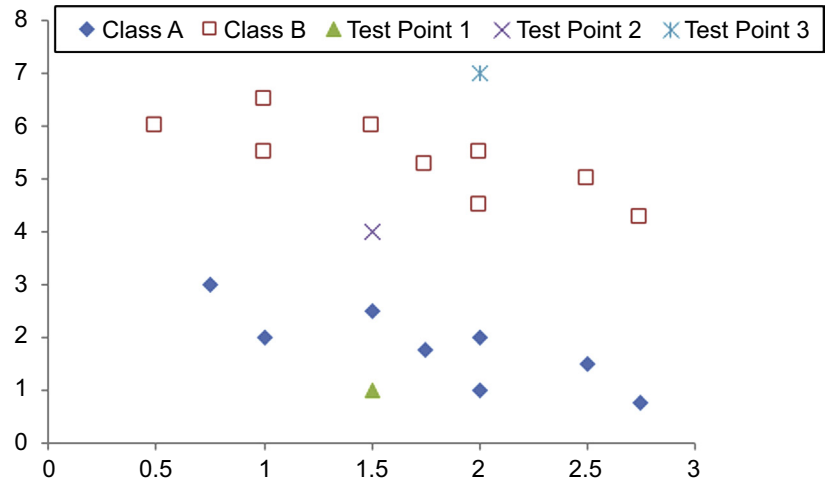



**FIGURE 4.52**
Two-class training data: class A (diamond) and class B (square). Points 1 to 3 are used to test the capability of the SVM.

**Table 4.11** A Simple Data Set to demonstrate SVM

| $x_1$ | $x_2$ | class |
|---|---|---|
| 1.5 | 2.5 | A |
| 2 | 2 | A |
| 1 | 2 | A |
| 0.75 | 3 | A |
| 2 | 1 | A |
| 1.75 | 1.75 | A |
| 2.75 | 0.75 | A |
| 2.5 | 1.5 | A |
| 0.5 | 6 | B |
| 1.5 | 6 | B |
| 2 | 5.5 | B |
| 1 | 5.5 | B |
| 1 | 6.5 | B |
| 2 | 4.5 | B |
| 1.75 | 5.25 | B |
| 2.75 | 4.25 | B |
| 2.5 | 5 | B |
| 1.5 | 1 | Test Point 1 |
| 1.5 | 4 | Test Point 2 |
| 2 | 7 | Test Point 3 |

shows the plot of the dataset. We will be using the model to classify the three test examples: (1.5, 1),(1.5, 4), and (2, 7).

**Step 1: Data preparation**

- Read simpleSVMdemo.csv into RapidMiner by either using the *Read csv* operator or import the data into your repository using Import csv file. The data set can be downloaded from the companion website www.LearnPredictiveAnalytics.com.
- Add a *Set Role* operator to indicate that *class* is a label attribute and connect it to the data retriever. See Figure 4.53a.

**Step 2: Modeling operator and parameters**

- In the Operators tab, type in **SVM**, drag and drop the operator into the main window, and connect it to Set Role. Leave the parameters of this operator in their default settings.
- Connect the "mod" output port of **SVM** to an *Apply Model* operator
- Insert a *Generate Data by User Specification* operator and click on the Edit List button of the attribute values parameter. When the dialog box opens up, click on Add Entry twice to create two test attribute names: $x_1$ and $x_2$. Set $x_1 = 2$ and $x_2 = 7$ under "attribute value." Note that you will need to change the attribute values for each new test point you want to classify.
- When this simple process is run, RapidMiner builds an SVM on the training data and applies the model to classify the test example, which was manually input using the Generate Data by User Specification operator.

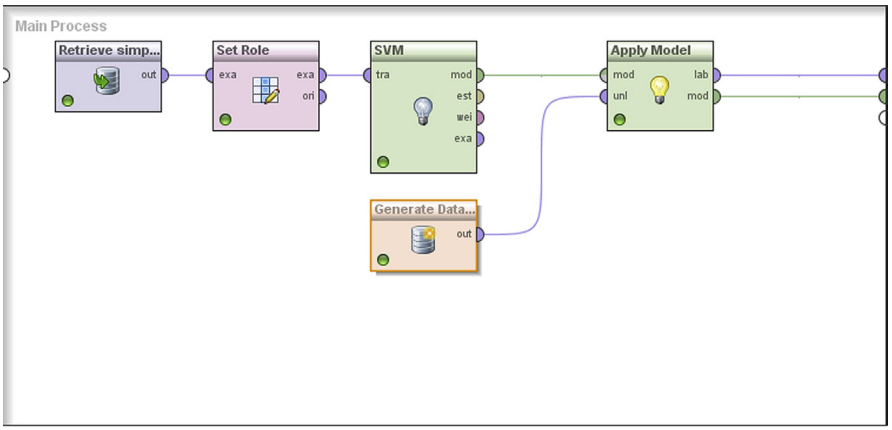



**FIGURE 4.53a**
A simple SVM setup for training and testing.

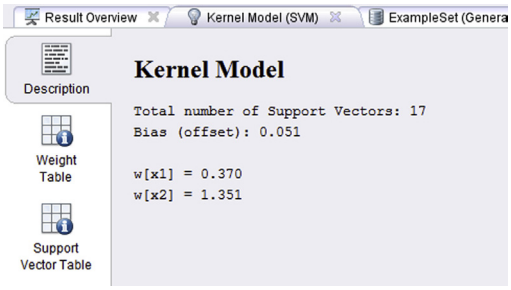



**FIGURE 4.53b**
The accompanying SVM model.

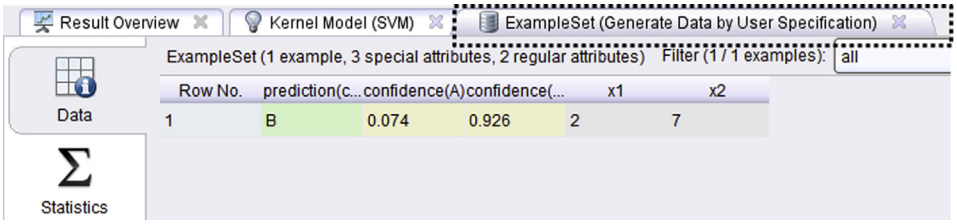



**FIGURE 4.54**
Applying the simple SVM to classify test point 1 from Figure 4.52.

**Step 3: Process execution and interpretation**
For more practical applications, we may not be very interested in the Kernel Model output; we show it here (Figure 4.53b) to observe what the hyperplane for this simplistic example looks like. Note that this is essentially the same form as Equation 4.19 with bias $b_0 = 0.051$, $w_1 = 0.370$, and $w_2 = 1.351$.

- The more interesting result in this case is the output from the "lab" port of the Apply Model, which is the result of applying the SVM model on the test point (2, 7).
- As you can see in Figure 4.54, the model has correctly classified this test point as belonging to class B (see the "prediction(class)" column). Furthermore, we are told that the confidence that this point belongs in class B is 92.6%. Looking at the chart, we see indeed that there is very little ambiguity about the classification of test point (2, 7).
- If you change the test example input to the point (1.5, 1), we see that this point would be classified under class A, with 88% confidence.
- However the same cannot be said of test point (1.5, 4); you can run the process and test for yourself!

In actual practice the labeled test data with prediction confidences are the most useful results from an SVM application.

### *Example 2: Applying SVM for a Linearly Nonseparable Example (Two Rings)*

The first example showed a linearly separable training data set. Suppose we want to now apply the same SVM (dot) kernel to the two ring problem we saw earlier in this chapter: what would the results look like? We know from looking at the data set that this is a nonlinear problem and the dot kernel should not work very well. We will confirm this intuitive understanding and demonstrate how it can be easily fixed in the steps described below.

**Step 1: Data preparation**

- Start a new process and read in the data set *nonlinearSVMdemodata.csv* using the same procedure as before. This dataset consists of 200 examples in four attributes: $x_1$, $x_2$, y, and ring. The ring is a binomial attribute with two values: inner and outer.
- Connect a *Set Role* operator to the data and select the "ring" variable to be the label.
- Connect a *Select Attributes* operator to this and select a subset of the attributes: $x_1$, $x_2$ and ring. Make sure that the *Include Special Attributes* checkbox is on.
- Connect a *Split Validation* operator. Set the "split" to relative, "split ratio" to 0.7, and "sampling type" to stratified.

**Step 2: Modeling operator and parameters**

- Double-click the *Split Validation* box and when you enter the nested layer, add an SVM operator in the training panel and *Apply Model* and *Performance (Classification)* operators in the testing panel.
- Once again, do not change the parameters for the SVM operator from its default values.
- Go back to the main level and add another *Apply Model* operator. Connect the "mod" output from the *Validation* box to the "mod" input port of *Apply Model (2)* and the "exa" output from the Validation box to the "unl" input port of *Apply Model (2)*. Also connect the "ave" output from the *Validation* box to the "res" port of the Main Process. Finally, connect the "lab" output from *Apply Model (2)* to the "res" port of the Main Process. Your final process should look like Figure 4.55.

**Step 3: Execution and interpretation**

- When you run this model, RapidMiner will generate two result tabs: *ExampleSet (Select Attributes)* and *PerformanceVector (Performance)*. Let's check the *performance* of the SVM classifier. Recall that 30% of the initial input examples are now going to be tested for classification accuracy (which is a total of 60 test samples).

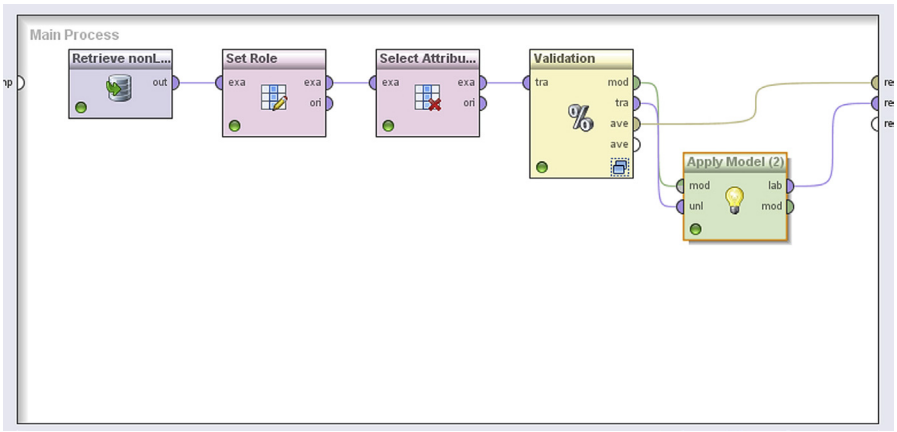



**FIGURE 4.55**
Setup for the nonlinear SVM demo model.

Table View  Plot View

**accuracy: 41.67%**

| | true inner | true outer | class precision |
|---|---|---|---|
| pred. inner | 12 | 17 | 41.38% |
| pred. outer | 18 | 13 | 41.94% |
| class recall | 40.00% | 43.33% | |

**FIGURE 4.56**
Prediction accuracy of a linear (dot) kernel SVM on nonlinear data.

- As you can see in Figure 4.56, the linear SVM can barely get 50% of the classes correct, which is to be expected considering we have linearly nonseparable data and we are using a linear (dot) kernel SVM.
- A better way to visualize this result is by means of the Scatter 3D Color plot. Click on the *ExampleSet (Select Attributes)* results tab and select Plot View and set up the Scatter 3D Color plot as shown in Figure 4.57.

The red-colored examples in the upper (outer) ring are correctly classified as belonging to the class "outer" while the cyan-colored examples have been incorrectly classified as belonging to class "inner." Similarly, the blue-colored examples in the lower (inner) ring have been correctly classified as belonging to class "inner" whereas the yellow-colored ones are not. As you can see, the classifier roughly gets about half the total number of test examples right.

To fix this situation, all we need to do is to go back to the SVM operator in the process and change the *kernel type* to *polynomial* (default degree 2.0) and rerun

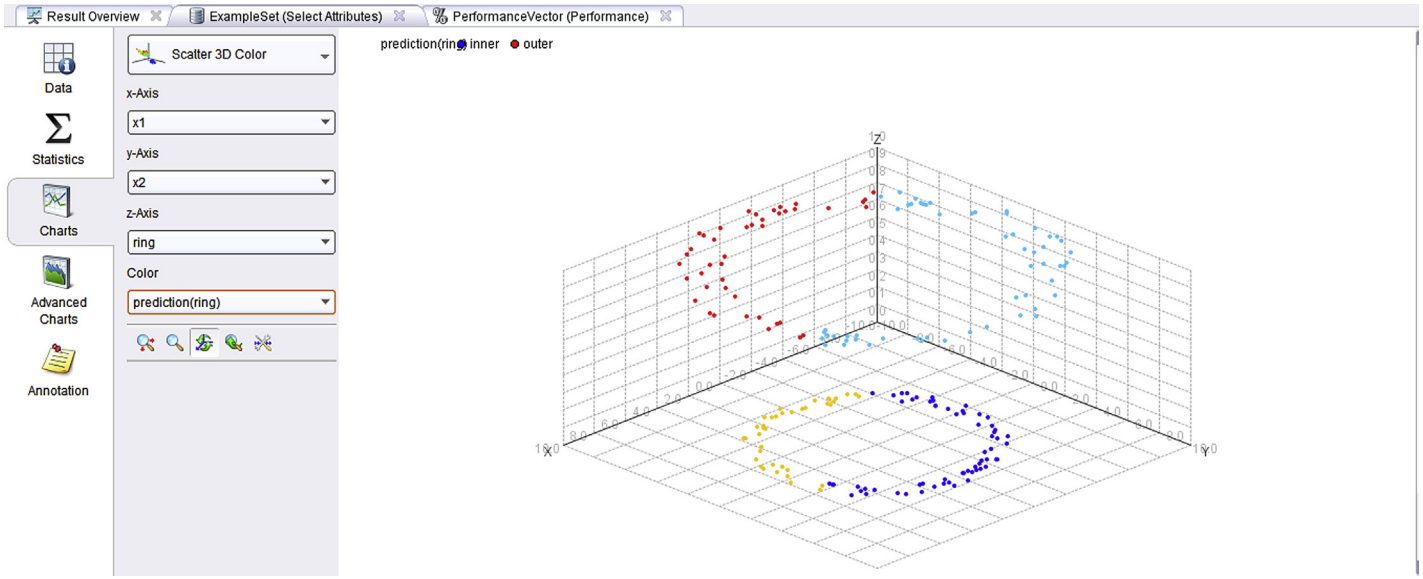



**FIGURE 4.57**
Visualizing the prediction from linear SVM.

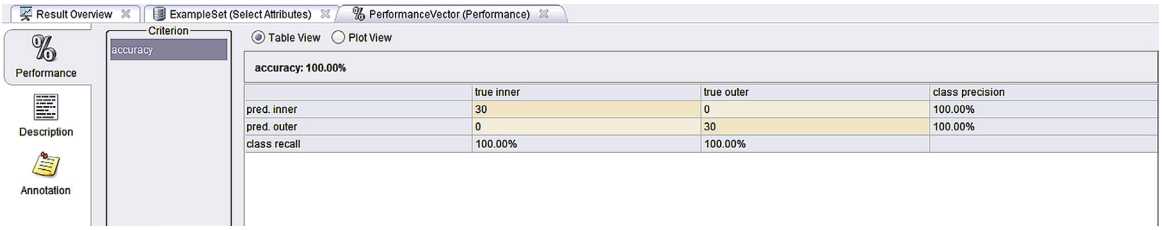



**FIGURE 4.58**
Classifying the two-ring nonlinear problem using a polynomial SVM kernel.

the analysis. This time we will be able to classify the points with 100% accuracy as seen in Figure 4.58. You will get the same result if you try a radial kernel as well.

The point of this exercise was to demonstrate the flexibility of SVMs and the ease of performing such trials using RapidMiner. Unfortunately with more realistic data sets, there is no way of knowing beforehand which kernel type would work best. The solution is to nest the SVM within an Optimization operator and explore a host of different kernel types and kernel parameters until we find one that performs reasonably well. (Optimization using RapidMiner is described in Chapter 13 Getting Started with RapidMiner.)

### *RapidMiner Parameter Settings*

There are many different parameters that can be adjusted depending upon the type of kernel function that is chosen. There is, however, one parameter that is very critical in optimizing SVM performance: this is the SVM complexity constant, C, which sets the penalties for misclassification, as was described in an earlier section. Most real world data sets are not cleanly separable and therefore will require the use of this factor. For initial trials, however, it is best to go with the default settings. The help in RapidMiner provides more details on how C impacts the performance and we will not repeat that here.

## 4.6.4 Conclusion

A disadvantage with higher order SVMs is the computational cost. In general since SVMs have to compute the dot product for every classification (and during training), very high dimensions or a large number of attributes can result in very slow computation times. However this disadvantage is offset by the fact that once an SVM model is built, small changes to the training data will not result in significant changes to the model coefficients as long as the support vectors do not change. This overfitting resistance is one of the reasons why SVMs have emerged as the most versatile of machine learning algorithms.

In summary, the key advantages of SVM are

- **Flexibility in application:** SVMs have been applied for activities from image processing to fraud detection to text mining.
- **Robustness:** Small changes in data does not require expensive remodeling.
- **Overfitting resistance:** The boundary of classes within data sets can be adequately described usually by only a few support vectors.

These advantages have to be balanced with the somewhat high computational costs of SVMs.

## 4.7 ENSEMBLE LEARNERS

In supervised data mining, the objective is to build a model that can explain the relationship between inputs and output. The model can be considered a hypothesis that can map new input data to predicted output. For a given training set, multiple hypotheses can explain the relationship with varying degrees of accuracy. While it is difficult to find the exact hypothesis from an infinite hypothesis space, we would like the modeling process to find the hypothesis that can *best* explain the relationship with least error.

Ensemble methods or learners optimize the hypothesis-finding problem by employing an array of individual prediction models and then combining them to form an aggregate hypothesis or model. These methods provide a technique for generating a better hypothesis by combining multiple hypotheses into one. Since a single hypothesis can be locally optimal or overfit a particular training set, combining multiple models can improve the accuracy by forcing a meta-hypothesis solution. It can be shown that in certain conditions this combined predictive power is better than the predictive power of individual models. Since different methods often capture different features of the solution space as part of any one model, the ensembles of models have emerged as the most important technique for many practical classification problems.

### 4.7.1 Wisdom of the Crowd

Ensemble models have a set of base models that accept the same inputs and predict the outcome individually. Then the outputs from all of these base models are combined, usually by voting, to form an ensemble output. This approach is similar to decision making by a committee or a board. The method of improving accuracy by drawing together the prediction of multiple models is also called *meta learning*. We see this similar decision-making methodology in higher courts of justice, corporate boards, and

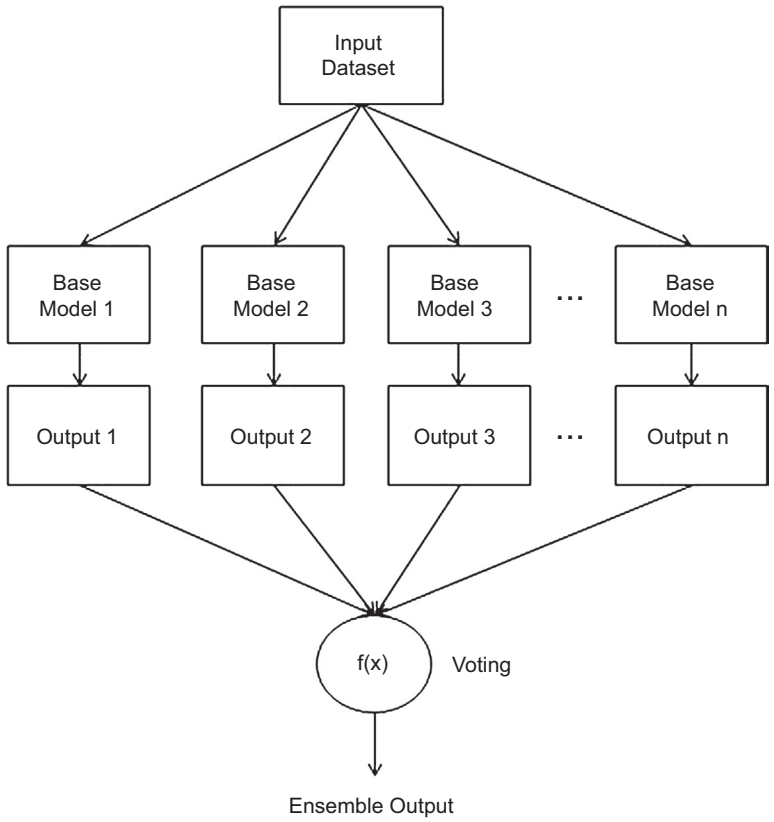



**FIGURE 4.59**
Ensemble model.

various committees in legislative bodies. While individual members have biases and options, the thinking here is collective decision making is better than one individual's assessment. Ensemble methods are used to improve the error rate and overcome the modeling bias of individual models. They can produce one strong learner by combining many weak learners. Figure 4.59 provides the framework of ensemble models.

The final step of aggregating the prediction is usually done by voting. The predicted class with more votes from the base learners is the output of the combined ensemble model. Base models predict the outcome with varied degrees of accuracy. Hence, we can also weight the vote by the accuracy rate of individual models, which causes base models with higher accuracy to have higher representation in the final aggregation than models with lower accuracy rate (Dietterich, 2007).

## PREDICTING DROUGHT

Drought is a period of time where a region experiences far less than average water supply. With the onset of climate change, there has been an increase in frequency and duration of drought conditions in many parts of the world. Immediate drought is caused by the development of high-pressure regions, which inhibits the formation of clouds, which results in low precipitation and lower humidity. Predicting drought conditions in a region is a very challenging task. There is no clear start and end point for draught duration. There are too many variables that impact the climate patterns that lead to drought conditions. Hence, there is no strong model to predict drought well ahead of time (Predicting Drought, 2013). Predicting drought seasons in advance would provide time for regional administrations to mitigate the consequences of the drought.

Droughts involve a myriad factors including groundwater level, air stream flow, soil moisture, topology, and large-scale global weather patterns like El Nino and La Nina (Patel, 2012). With thousands of attributes and many unknown variables that influence the conditions for drought, there is no "silver bullet" massive model for predicting when drought is going to hit a region with a high degree of accuracy. What we have is many different "weak" models that use some of the thousands of attributes available, which make predictions marginally better than pure chance. These weak models may provide different drought predictions for the same region and time, based on the diverse input variables for each model. We can summarize the prediction by combining the predictions of individual models and take a vote. Ensemble models provide a systemic method to combine many weak models into one better model. Most of the data mining models deployed in production applications are ensemble models. Ensemble models greatly reduce generalization errors and improve the accuracy of the overall prediction, if certain conditions are met.

### 4.7.2 How it Works

Let's take an example of a hypothetical corporate boardroom with three board members. Assume that individually each board member makes wrong decisions about 20% of time. The board needs to make a yes/no decision for a major project proposal. If all board members make consistent unanimous decision every time, then the error rate of the board as a whole is 20%. But, if each board member's decisions are *independent* and if their outcomes are not correlated, the board makes an error only when more than *two board members make an error* at the same time. The board makes an error only when the majority of its members make an error. We can calculate the error rate of the board using the binomial distribution.

In binomial distribution, the probability of k successes in n independent trials each with a success rate of p is given by a probability mass function:

$$P(k) = \left(\frac{n}{k}\right) p^k (1-p)^{n-k} \tag{4.22}$$

P(Board wrong) = P($k \geq 2$) = P(2 members wrong) + P(3 members wrong)

$$P(\text{Board wrong}) = \binom{n}{3} p^k (1-p)^{n-k} + \binom{n}{2} p^k (1-p)^{n-k}$$

$$= \binom{3}{3} 0.2^3 (1-0.2)^0 + \binom{3}{2} 0.2^2 (1-0.2)^1$$

$$= 0.008 + 0.096$$

$$= 0.104$$

$$= 10.4\%$$

In this example, the error rate of the board (10.4%) is *less* than the error rate of the individuals (20%)! We therefore see the impact of collective decision making. A generic formula for calculating error rate for the ensemble is given by

$$P(\text{ensemble wrong}) = P(k \geq \text{round}(n/2)) = \sum_{k=0}^{n} \binom{n}{k} p^k (1-p)^{n-k}$$

However, some important criteria to note are:

- Each member of the ensemble should be independent.
- The individual model error rate should be less than 50% for binary classifiers.

If the error rate of the base classifier is more than 50%, its prediction power is worse than pure chance and hence, it is not a good model to begin with. Achieving the first criterion of independence amongst the base classifier is difficult. However there are a few techniques available to make base models as diverse as possible. In the board analogy, having a board with diverse and independent members makes statistical sense. Of course, they all have to make right decisions more than half the time.

### *Achieving the Conditions for Ensemble Modeling*

We will be able to take advantage of combined decision-making power only if the base models are good to begin with. While meta learners can form a strong learner from several weak learners, those weak learners should be better than random guessing. Because all the models are developed based on the same training set, the diversity and independence condition of the model is difficult to accomplish. While complete independence of the base models cannot be achieved, we can take steps to promote independence by changing the training sets for each base model, varying the input attributes, building different classes of modeling techniques and algorithms, and changing the modeling parameters to build the base models. To achieve the diversity in the

base models, we can alter the conditions in which the base model is built. The most commonly used conditions are:

- **Different model algorithms**: The same training set can be used to build different classifiers, such as decision trees using multiple algorithms, naïve Bayesian, k-nearest neighbors, artificial neural networks, etc. The inherent characteristics of these models will be different, which yields different error rates and a diverse base model set.
- **Parameters within the models:** Changing the parameters like depth of the tree, gain ratio, and maximum split for decision tree model can produce multiple decision trees. The same training set can be used to build all the base models.
- **Changing the training record set:** Since the training data is the key contributor to the error in a model, changing the training set to build the base model is one effective method for building multiple independent base models. A training set can be divided into multiple sets and each set can be used to build one base model. However, this technique requires a sufficiently large training set and is seldom used. Instead, we can sample training data with replacement from a data set and repeat the same process for other base models.
- **Changing the attribute set:** Similar to changing the training data where a sample of records are used for the building of each base model, we can sample the attributes for each base model. This technique works if the training data have a large number of attributes.

In the next few sections, we will be reviewing specific approaches to building ensemble models based on the above techniques on promoting independence among base models. There are some limitations in using ensemble models. If different algorithms are used for the base models, they impose different restrictions on the type of input data that can be used. Hence it could create a superset of restrictions to inputs for an ensemble model.

### 4.7.3 How to Implement

In data mining tools, ensemble modeling operators can be found in meta learning or ensemble learning groupings. In RapidMiner, since ensemble modeling is used in the context of predicting, all the operators are located in Modeling > Classification and Regression > Meta Modeling. The process of building ensemble models is very similar to that of building any classification models like decision tress or neural networks. Please refer to previous classification algorithms for steps to develop individual classification processes and models in RapidMiner. There are a few options to choose for implementing meta modeling in RapidMiner. In the next few pages we will review the implementation of ensemble modeling with simple voting and a couple of other techniques to make the base models independent by altering examples for the training set.

### *Ensemble by Voting*

Implementing an ensemble classifier starts with building a simple base classification process. For this example, we can build a decision tree process with the Iris data set as shown in Section 4.1 Decision Trees. The standard decision tree process involves data retrieval and a decision tree model, followed by applying the model to an unseen test data set sourced from the Iris data set and using a performance evaluation operator. To make it an ensemble model, the *Decision Tree* operator has to be replaced with the *Vote* operator from the meta learning folder. All other operators will remain the same. The ensemble process will look similar to the process shown in Figure 4.60.

The *Vote* operator is an ensemble learner that houses multiple base models inside the *inner subprocess* (Mierswa et al., 2006). The model output from the vote process behaves like any other classification model and it can be applied in any scenario where a decision tree can be used. In the apply model phase, the predicted classes are tallied up amongst all the base classifiers and the class with highest number of votes is the predicted class for the ensemble model.

On double-clicking the nested *Vote* meta modeling operator, we can add multiple base classification models inside the *Vote* operator. All these models accept the same training set and provide an individual base model as output. In this example we have added three models: decision tree, k-NN and naïve Bayes. Figure 4.61 shows the inner subprocess of the *Vote* meta modeling operator. The act of tallying

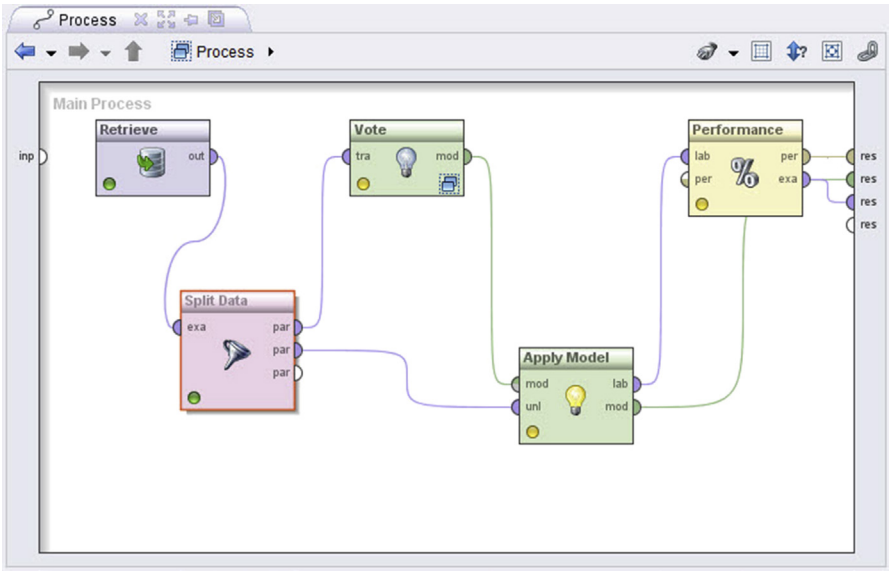



**FIGURE 4.60**

Data mining process using ensemble model.

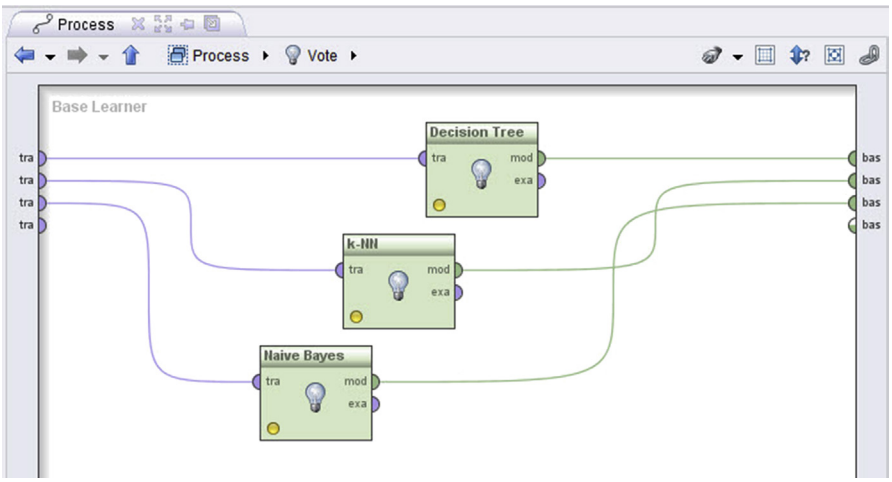



**FIGURE 4.61**
Subprocess inside the Vote operator.

all the predictions of these base learners and providing the majority prediction is the job of the meta model—the *Vote* modeling operator. This is the aggregation step in ensemble modeling and in RapidMiner it is called a stacking model. A stacking model is built into the *Vote* operator and is not visible on the screen.

The ensemble process with the *Vote* meta model can be saved and executed. Once the process is executed, the output panel of the performance vector is no different than a normal performance vector. Since this process has a meta model, the model panel in the results window exhibits new information, as shown in Figure 4.62. The model window shows all the individual base models and one stacking model, which is the combination of all the base models. The *Vote* meta model is simple to use wherever an individual base model could have been used independently. The limitation of the model is that all the base learners use the same training data set and different base models impose restrictions on what data types they can accept.

### *Bootstrap Aggregating or Bagging*

Bagging is a technique where base models are developed by changing the training set for every base model. In a given training set T of n records, m training sets are developed each with n records, by sampling with replacement. Each training set $T_1$, $T_2$, $T_3$, …, $T_m$ will have the same record count of n as the original training set T. Because they are sampled with replacement, they can contain duplicate records. This is called *bootstrapping*. Each sampled training set is then used for a base model preparation. Through bootstrapping, we have a set of m base models and the prediction of each model is aggregated for an ensemble model. This combination of bootstrapping and aggregating is called *bagging*.

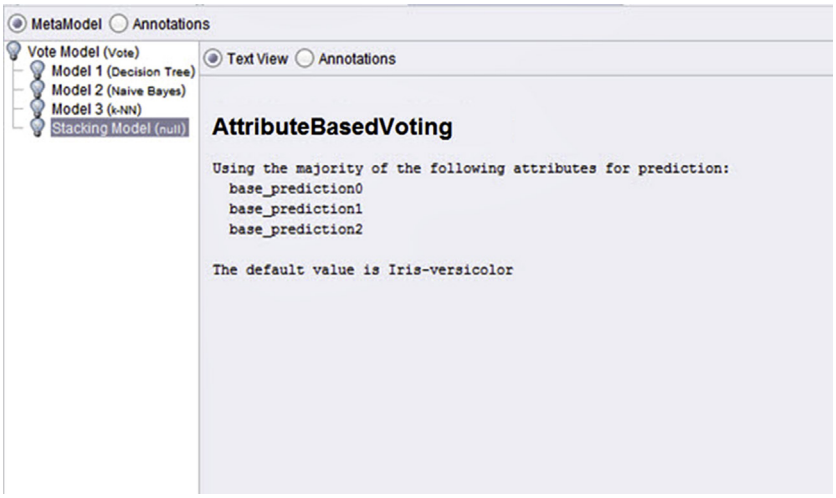



**FIGURE 4.62**
Output of ensemble model based on voting.

On average, each base training set $T_i$ contains about 63% unique training records as compared to the original training set T. Sampling with replacement of n records contains $1 - (1 - 1/n)^n$ unique records. When n is sufficiently large we get $1 - 1/e = 63.2\%$ unique records on average. The rest of the data contains duplicates from already sampled data. The process of bagging improves the stability of unstable models. Unstable models like decision trees and neural network are highly dependent even on slight changes in the training data. Because a bagging ensemble combines multiple hypotheses of the same data, the new aggregate hypothesis helps neutralize these training data variations.

### *Implementation*

The *Bagging* operator is available in the meta learning folder: Modeling > Classification and Regression > Meta modeling > Bagging. Like the *Vote* meta operator, *Bagging* is a nested operator with an inner subprocess. Unlike the vote process, bagging has only one model in the inner subprocess. Multiple base models are generated by changing the training data set. The *Bagging* operator has two parameters.

- **Sample ratio:** Indicates the fraction of records used for training.
- **Iterations (m):** Number of base modes that need to be generated.

Figure 4.63 shows the RapidMiner process for the *Bagging* operator. Figure 4.64 shows the inner subprocess for the *Bagging* operator with one model specification. Internally multiple base models are generated based on iterations (m) configured in the Bagging parameter. In this example, we are using a decision tree model for the inner subprocess.

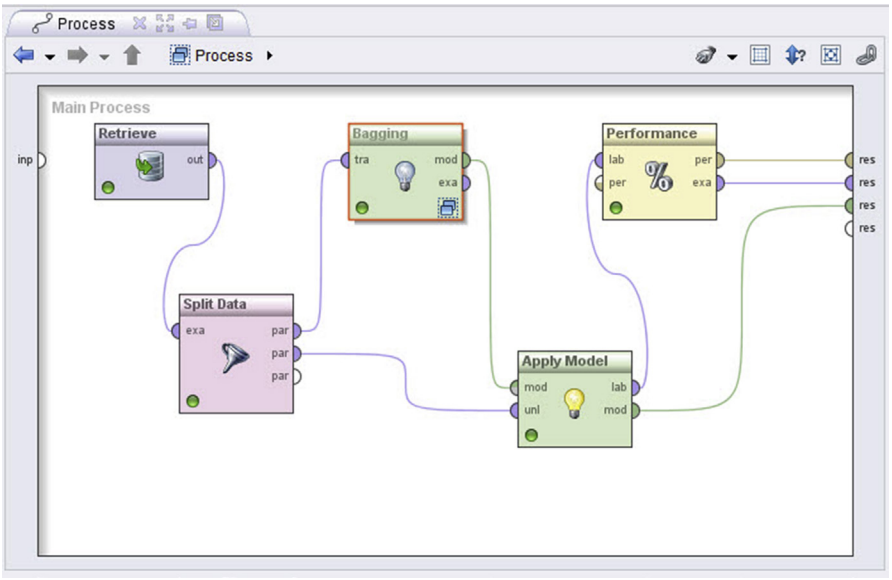



**FIGURE 4.63**
Ensemble process using bagging.

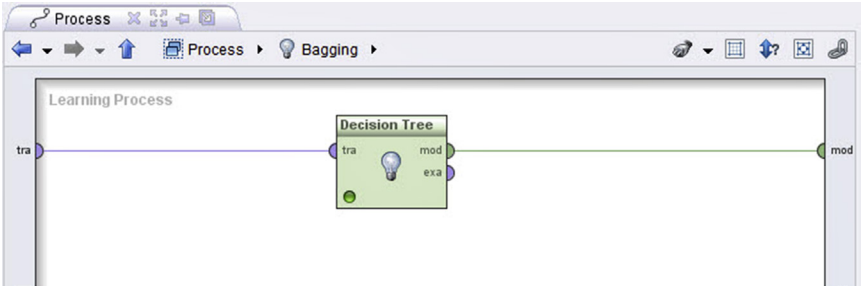



**FIGURE 4.64**
Bagging subprocess.

The RapidMiner process for bagging can be saved and executed. Similar to the *Vote* meta model, the *Bagging* meta model acts as one model with multiple base models inside. The results window shows the practiced example set, performance vector, and bagging model description. In the results window, we can examine all *m* (in this case 10) models that are developed based on *m* iterations of the training set. The base model results are aggregated using simple voting. Bagging is particularly useful when there is anomaly in the training data set that impacts the individual model significantly. Bagging provides a useful framework where the same data mining algorithm is used for all base

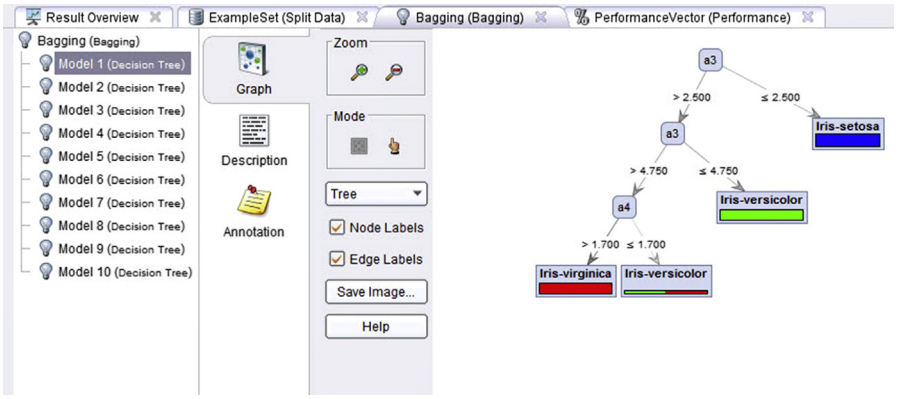



**FIGURE 4.65**
Output of bagging models.

learners. However, each base model differs because the training data used by the base learners are different. Since each base model explores a different solution space, the performance of the ensemble model would be better than that of base models. Figure 4.65 shows the model output of *Bagging* meta model with constituent decision trees.

### *Boosting*

Boosting offers another approach to building an ensemble model by manipulating training data similar to bagging. As with bagging, it provides a solution to combine many weak learners into one strong learner, by minimizing bias or variance due to training records. Unlike bagging, boosting trains the base models in sequence one by one and assigns weights for all training records. The boosting process concentrates on the training records that are hard to classify and overrepresents them in the training set for the next iteration.

The boosting model is built by an iterative and sequential process where a base model is built and tested with all of the training data, and based on the outcome, the next base model is developed. To start with, all training records have equal weight. The weight of the record is used for the sampling distribution for selection with replacement. A training sample is selected based on the weights and used for model building. Then the model is used for testing with the whole training set. Incorrectly classified records are assigned a higher weight and correctly classified records are assigned a low weight, so hard-to-classify records have a higher propensity of selection for the next round. The training sample for the next round will be most likely filled with incorrectly classified records from the previous round. Hence the next model will focus on the hard-to-classify data space.

Boosting assigns the weight for each training record and has to adaptively change the weight based on difficulty of classification. This results in an ensemble of base learners specialized in classifying both easy-to-classify and hard-to-classify records. When applying the model, all base learners are combined through a simple voting aggregation.

### *AdaBoost*

AdaBoost is one of the most popular implementations of the boosting ensemble approach. It is adaptive because it assigns weights for base models ($\alpha$) based on the accuracy of the model, and changes weights of the training records ($w$) based on the accuracy of the prediction. Here is the framework of the AdaBoost ensemble model with m base classifiers and n training records ($(x_1,y_1)$, $(x_2,y_2)$, …, $(x_n,y_n)$). Following are the steps involved in AdaBoost:

1. Each training record is assigned an uniform weight $w_i = 1/n$.
2. Training records are sampled and the first base classifier $b_k(x)$ is built.
3. The error rate for the base classifier can be calculated by Equation 4.23:

$$e_k = \sum_{k=1}^{n} w_i * I(b_k(x_i) \neq y_i) \tag{4.23}$$

where I(x) = 1 when the prediction is right and 0 when the prediction is incorrect.

4. The weight of the classifier can be calculated as $\alpha_k = \ln(1 - e_k)/e_k$. If the model has a low error rate, then the weight of the classifier is high and vice versa.
5. Next, the weights of all training records are updated by

$$w_{k+1}(i+1) = w_k(i) * e^{(\alpha_k F(bk(xi) \neq yi))}$$

where F(x) = –1 if the prediction is right and F(x) = 1 if the prediction is wrong.

Hence, the AdaBoost model updates the weights based on the prediction and the error rate of the base classifier. If the error rate is more than 50%, the record weight is not updated and reverted back to the next round.

### *AdaBoost Model in RapidMiner*

The *AdaBoost* operator is available in the meta learning folder: Modeling > Classification and Regression > Meta modeling > AdaBoost. The operator functions similar to Bagging and has an inner subprocess. The number of iterations or base models is a configurable parameter for the *AdaBoost* operator. Figure 4.66 shows the AdaBoost data mining process. This example uses the Iris data set with the *Split Data* operator for generating training and test data sets. The output of the AdaBoost model is applied to the test set and the performance is evaluated by the *Performance* operator.

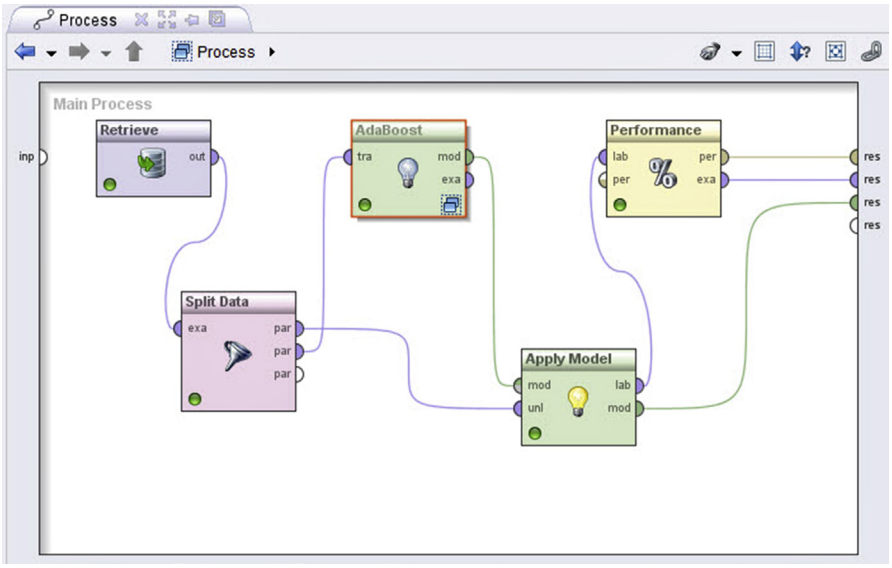



**FIGURE 4.66**
Data mining process using AdaBoost.

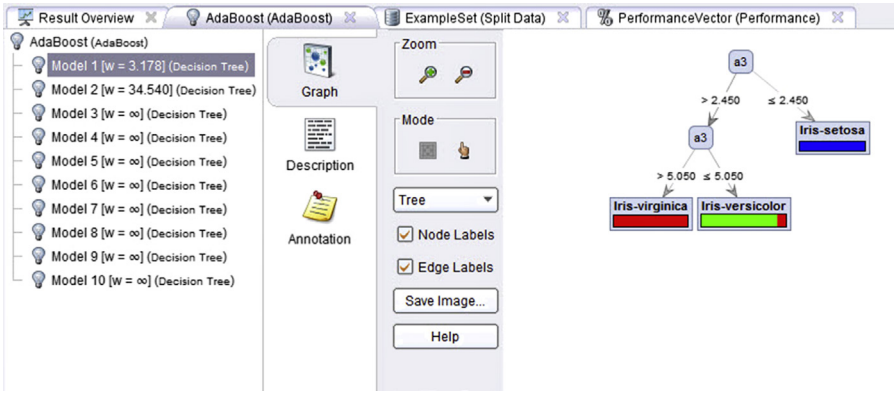



**FIGURE 4.67**
Output of AdaBoost model.

The number of iterations used in the AdaBoost is three, which is specified in the parameter. In the inner process, the model type can be specified. In this example the decision tree model is used. The completed RapidMiner process is saved and executed. The result window has the output ensemble model, base models, and the predicted records. The model window shows the decision trees for the base classifiers. Figure 4.67 shows the result output for the AdaBoost model.

### Random Forest

Recall that in the bagging technique, for every iteration, a sample of training records is considered for building the model. The random forest technique uses a similar concept to the one used in bagging. When deciding on splitting each node in a decision tree, the random forest only considers a random subset of all the attributes in the training set. To reduce the generalization error, the algorithm is randomized in two levels, training record selection and attribute selection, in the inner working of each base classifier. The random forests concept was first put forward by Leo Breiman and Adele Cutler (Breiman, 2001).

In general, the model works using the following steps. If there are n training records with m attributes, and let k be the number of trees in the forest; then for each tree:

1. An n random sample is selected with replacement. This step is similar to bagging.
2. A number D is selected, where D << m. D determines the number of attributes to be considered for node splitting.
3. A decision tree is started. For each node, instead of considering all m attributes for the best split, a random number D attributes are considered. This step is repeated for every node.
4. As in any ensemble, the greater the diversity of the base trees, the lower the error of the ensemble.

Once all the trees in the forest are built, for every new record, all the trees predict a class and vote for the class with equal weights. The most predicted class by the base trees is the prediction of the forest (Gashler et al., 2008).

### Implementation

The *Random Forest* operator is available in Modeling > Classification and Regression > Tree Induction > Random Forest. It works similarly to the other ensemble models where the user needs to specify the number of base trees to be built. Since the inner base model is always a decision tree, there is no explicit inner subprocess specification. Bagging or boosting ensemble models require explicit inner subprocess specification. All the tree-specific parameters like leaf size, depth, and split criterion can be specified in the *Random Forest* operator. The key parameter that specifies the number of base trees is *Number of Trees*. Figure 4.68 shows the RapidMiner process with the Iris data set, the *Random Forest* modeling operator, and the *Apply Model* operator. For this example, the number of base trees is specified as 10. The process looks and functions similarly to a simple decision tree classifier.

Once the process is executed, the results window shows the model, predicted output, and performance vector. Similar to other meta model output, the

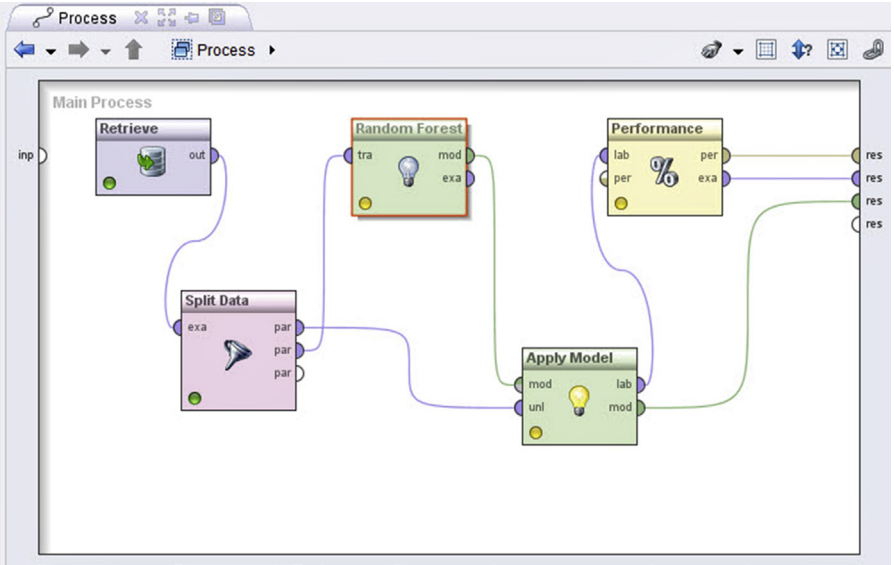



**FIGURE 4.68**
Data mining process using the Random Forest operator.

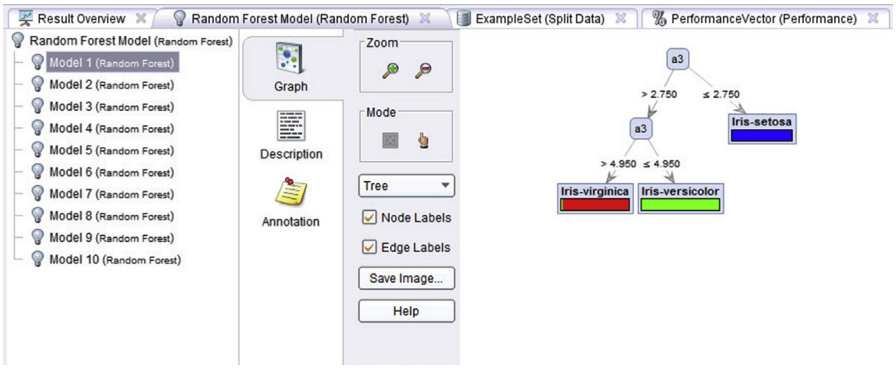



**FIGURE 4.69**
Output of Random Forest models.

Random Forest model shows the trees for all base classifiers. Figure 4.69 shows the model output for the *Random Forest* operator. Notice that the nodes are different in each tree. Since the attribute selection for each node is randomized, each base tree is different. Thus the Random Forest models strive to reduce the generalization error of the decision tree model. The Random Forest models are very useful as baseline ensemble models for comparative purposes.

### 4.7.4 Conclusion

Most of the data mining models developed for production applications are built on ensemble models. They are used in wide range of applications, including political forecasting (Montgomery et al., 2012), weather pattern modeling, media recommendation, web page ranking (Baradaran Hashemi et al., 2010), etc. Since many algorithms approach the problem of modeling the relationship between input and output differently, it makes sense to aggregate the prediction of a diverse set of approaches. Ensemble modeling reduces the generalization error that arises due to overfitting the training data set. The four ensemble techniques discussed provide fundamental methods of developing a cohort of base models by choosing different algorithms, changing parameters, changing training records, sampling, and changing attributes. All these techniques can be combined in one ensemble model. There is no one approach for ensemble modeling; all the techniques discussed in this chapter were proven to perform better than base models as long as they are diverse (Polikar, 2006). The wisdom of crowds makes sense in data mining as long as "group thinking" is controlled by promoting independence amongst base models.

## REFERENCES


Altman, N. S. (1992). An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression. *The American Statistician*, *46*(3), 175–185.

Baradaran Hashemi, H., Yazdani, N., Shakery, A., & Pakdaman Naeini, M. (2010). Application of ensemble models in web ranking. *2010 5th International Symposium on Telecommunications*, 726–731. http://dx.doi.org/10.1109/ISTEL.2010.5734118.

Breiman, L. (2001). Random Forests. *Machine Learning*, *45*, 5–32.

Brieman, L. F. (1984). *Classification and Regression Trees*. Chapman and Hall.

Cohen, W. W. (1995). Fast Effective Rule Induction. *Machine Learning: Proceedings of the Twelfth International Conference*.

Cortes, C. A. (1995). Support Vector Networks. *Machine Learning*, 273–297.

Cover, T. A. (1991). Entropy, Relative Information, and Mutual Information. In T. A. Cover (Ed.), *Elements of Information Theory* (pp. 12–49). John Wiley and Sons.

Dietterich, T. G. (2007). *Ensemble Methods in Machine Learning*. Retrieved from http://www.eecs.wsu.edu/~holder/courses/CptS570/fall07/papers/Dietterich00.pdf.

Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of Human Genetics*, *7*, 179–188. http://dx.doi.org/10.1111/j.1469-1809.1936.tb02137.x.

Fletcher, R. (1987). *Practical Methods of Optimization*. New York: John Wiley.

Gashler, M., Giraud-Carrier, C., & Martinez, T. (2008). Decision Tree Ensemble: Small Heterogeneous Is Better Than Large Homogeneous. *2008 Seventh International Conference on Machine Learning and Applications*, 900–905. http://dx.doi.org/10.1109/ICMLA.2008.154.

Grabusts, P. (2011). The Choice of Metrics for Clustering Algorithms. *Proceedings of the 8th International Scientific and Practical Conference*, *II*(1), 70–76.

Haapanen, R., Lehtinen, K., Miettinen, J., Bauer, M. E., & Ek, A. R. (2001). Progress in adapting k-NN methods for forest mapping and estimation using the new annual Forest Inventory and Analysis data. In Third Annual Forest Inventory and Analysis Symposium (p. 87).

Hill, T., & Lewicki, P. (2007). *Statistics: Methods and Applications* p. 815.

Hsu, C.-W. C.-C.-J. (2003). A practical guide to support vector classification. *Taipei: Department of Computer Science*, National Taiwan University.

Laine, A. (2003). *Neural Networks. In Encyclopedia of Computer Science 4th edition*. John Wiley and Sons Ltd. 1233–1239.

Langley, P., & Simon, H. a. (1995). Applications of machine learning and rule induction. *Communications of the ACM*, *38*(11), 54–64. http://dx.doi.org/10.1145/219717.219768.

Peterson, L. (2009). K-Nearest Neighbors. *Scholarpedia*. Retrieved from http://www.scholarpedia.org/article/K-nearest_neighbor.

Li, E. Y. (1994). Artificial neural networks and their business applications. *Information & Management*, *27*(5), 303–313. http://dx.doi.org/10.1016/0378-7206(94)90024-8.

Matan, O., et al. (1990). Handwritten Character Recognition Using Neural Network Architectures. *4th USPS Advanced Technology Conference*, 1003–1011.

McInerney, D. (2005). Remote Sensing Applications k-NN Classification. *Remote Sensing Workshop*. Retrieved April 27, 2014, from http://www.forestry.gov.uk/pdf/DanielMcInerneyworkshop.pdf/$FILE/DanielMcInerneyworkshop.pdf.

Meagher, P. (2005). *Calculating Entropy for Data Mining*. Retrieved from O'Reilly OnLamp.com PHP Dev Center. http://www.onlamp.com/pub/a/php/2005/01/06/entropy.html?page=1.

Mierswa, I., Wurst, M., Klinkenberg, R., Scholz, M., & Euler, T. (2006). YALE: Rapid prototyping for complex Data Mining tasks. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Vol. 2006). 935–940. http://dx.doi.org/10.1145/1150402.1150531.

Montgomery, J.M., Hollenbach, F.M., Ward. M.D. (2012). Improving predictions using ensemble Bayesian model averaging. *Political Analysis*, *20*(3): 271–291.

Patel, P. (2012). Predicting the Future of Drought Prediction. *IEEE Spectrum*. Retrieved April 26, 2014, from http://spectrum.ieee.org/energy/environment/predicting-the-future-of-drought-prediction.

Polikar, R. (2006). Ensemble Based Systems in Decision Making. *IEEE CIRCUITS AND SYSTEMS MAGAZINE*, 21–45.

Predicting Drought (2013). National Drought Mitigation Center. Retrieved April 26, 2014, from http://drought.unl.edu/DroughtBasics/PredictingDrought.aspx.

Prosess Software (2013). Introduction to Bayseian Filtering. PreciseMail Whitepapers, 1–8. Retrieved from www.process.com.

Quinlan, J. (1986). Induction of Decision Trees. *Machine Learning*, 81–106.

Rish, I. (2001). An empirical study of the naive Bayes classifier. *IBM Research Report*.

Sahami, M., Dumais, S., Heckerman, D., & Horvitz, E. (1998). A Bayesian Approach to Filtering Junk E-Mail. Learning for Text Categorization. Papers from the 1998 Workshop. *AAAI Technical Report*.

Saian, R., & Ku-Mahamud, K. R. (2011). Hybrid Ant Colony Optimization and Simulated Annealing for Rule Induction. *2011 UKSim 5th European Symposium on Computer Modeling and Simulation*, 70–75. http://dx.doi.org/10.1109/EMS.2011.17.

Shannon, C. (1948). A Mathematical Theory of Communication. *Bell Systems Technical Journal*, 379–423.

Smola, A. A. (2004). A tutorial on support vector regression. *Statistics and Computing*, 199–222.

Tan, P.-N., Michael, S., & Kumar, V. (2005). Classfication and Classification: Alternative Techniques. In P.-N. Tan, S. Michael, & V. Kumar (Eds.), *Introduction to Data Mining* (pp. 145–315). Boston, MA: Addison-Wesley.

Zdziarski, J. A. (2005). *Ending spam: Bayesian content filtering and the art of statistical language classification*. No Starch Press.