WEKA

Machine Learning Algorithms in Java

Ian H. Witten
Department of Computer Science
University of Waikato
Hamilton, New Zealand
E-mail: ihw@cs.waikato.ac.nz

Eibe Frank
Department of Computer Science
University of Waikato
Hamilton, New Zealand
E-mail: eibe@cs.waikato.ac.nz

This tutorial is Chapter 8 of the book *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Cross-references are to other sections of that book.

Nuts and bolts: Machine learning algorithms in Java

All the algorithms discussed in this book have been implemented and made freely available on the World Wide Web (*www.cs.waikato.ac.nz/ml/weka*) for you to experiment with. This will allow you to learn more about how they work and what they do. The implementations are part of a system called Weka, developed at the University of Waikato in New Zealand. "Weka" stands for the Waikato Environment for Knowledge Analysis. (Also, the *weka*, pronounced to rhyme with *Mecca*, is a flightless bird with an inquisitive nature found only on the islands of New Zealand.) The system is written in Java, an object-oriented programming language that is widely available for all major computer platforms, and Weka has been tested under Linux, Windows, and Macintosh operating systems. Java allows us to provide a uniform interface to many different learning algorithms, along with methods for pre- and postprocessing and for evaluating the result of learning schemes on any given dataset. The interface is described in this chapter.

There are several different levels at which Weka can be used. First of all, it provides implementations of state-of-the-art learning algorithms that you can apply to your dataset from the command line. It also includes a variety of tools for transforming datasets, like the algorithms for discretization

discussed in Chapter 7. You can preprocess a dataset, feed it into a learning scheme, and analyze the resulting classifier and its performance—all without writing any program code at all. As an example to get you started, we will explain how to transform a spreadsheet into a dataset with the right format for this process, and how to build a decision tree from it.

Learning how to build decision trees is just the beginning: there are many other algorithms to explore. The most important resource for navigating through the software is the online documentation, which has been automatically generated from the source code and concisely reflects its structure. We will explain how to use this documentation and identify Weka's major building blocks, highlighting which parts contain supervised learning methods, which contain tools for data preprocessing, and which contain methods for other learning schemes. The online documentation is very helpful even if you do no more than process datasets from the command line, because it is the only complete list of available algorithms. Weka is continually growing, and—being generated automatically from the source code—the online documentation is always up to date. Moreover, it becomes essential if you want to proceed to the next level and access the library from your own Java programs, or to write and test learning schemes of your own.

One way of using Weka is to apply a learning method to a dataset and analyze its output to extract information about the data. Another is to apply several learners and compare their performance in order to choose one for prediction. The learning methods are called *classifiers.* They all have the same command-line interface, and there is a set of generic command-line options—as well as some scheme-specific ones. The performance of all classifiers is measured by a common evaluation module. We explain the command-line options and show how to interpret the output of the evaluation procedure. We describe the output of decision and model trees. We include a list of the major learning schemes and their most important scheme-specific options. In addition, we show you how to test the capabilities of a particular learning scheme, and how to obtain a bias-variance decomposition of its performance on any given dataset.

Implementations of actual learning schemes are the most valuable resource that Weka provides. But tools for preprocessing the data, called *filters,* come a close second. Like classifiers, filters have a standardized command-line interface, and there is a basic set of command-line options that they all have in common. We will show how different filters can be used, list the filter algorithms, and describe their scheme-specific options.

The main focus of Weka is on classifier and filter algorithms. However, it also includes implementations of algorithms for learning association rules and for clustering data for which no class value is specified. We briefly discuss how to use these implementations, and point out their limitations.

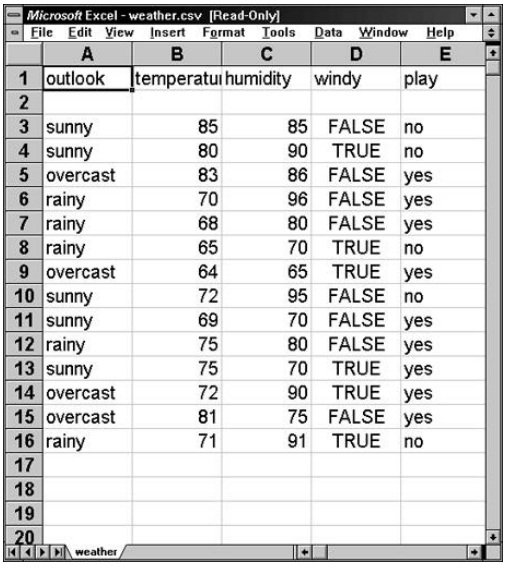
In most data mining applications, the machine learning component is just a small part of a far larger software system. If you intend to write a data mining application, you will want to access the programs in Weka from inside your own code. By doing so, you can solve the machine learning subproblem of your application with a minimum of additional programming. We show you how to do that by presenting an example of a simple data mining application in Java. This will enable you to become familiar with the basic data structures in Weka, representing instances, classifiers, and filters.

If you intend to become an expert in machine learning algorithms (or, indeed, if you already are one), you'll probably want to implement your own algorithms without having to address such mundane details as reading the data from a file, implementing filtering algorithms, or providing code to evaluate the results. If so, we have good news for you: Weka already includes all this. In order to make full use of it, you must become acquainted with the basic data structures. To help you reach this point, we discuss these structures in more detail and explain example implementations of a classifier and a filter.

8.1 Getting started

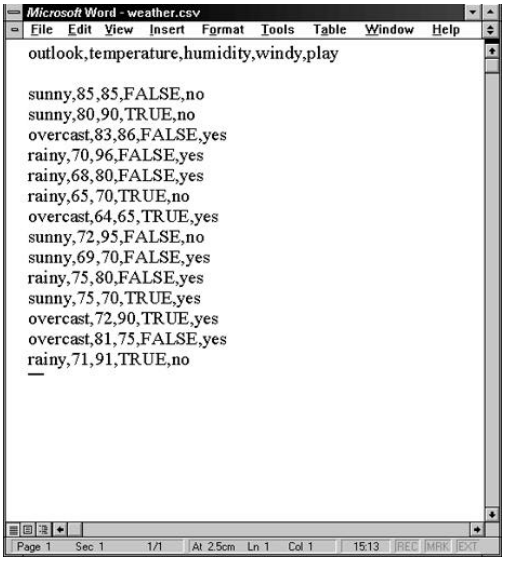
Suppose you have some data and you want to build a decision tree from it. A common situation is for the data to be stored in a spreadsheet or database. However, Weka expects it to be in ARFF format, introduced in Section 2.4, because it is necessary to have type information about each attribute which cannot be automatically deduced from the attribute values. Before you can apply any algorithm to your data, is must be converted to ARFF form. This can be done very easily. Recall that the bulk of an ARFF file consists of a list of all the instances, with the attribute values for each instance being separated by commas (Figure 2.2). Most spreadsheet and database programs allow you to export your data into a file in comma-separated format—as a list of records where the items are separated by commas. Once this has been done, you need only load the file into a text editor or a word processor; add the dataset's name using the `@relation` tag, the attribute information using `@attribute`, and a `@data` line; save the file as raw text—and you're done!

In the following example we assume that your data is stored in a Microsoft Excel spreadsheet, and you're using Microsoft Word for text processing. Of course, the process of converting data into ARFF format is very similar for other software packages. Figure 8.1a shows an Excel spreadsheet containing the weather data from Section 1.2. It is easy to save this data in comma-separated format. First, select the *Save As…* item from the *File* pull-down menu. Then, in the ensuing dialog box, select *CSV*

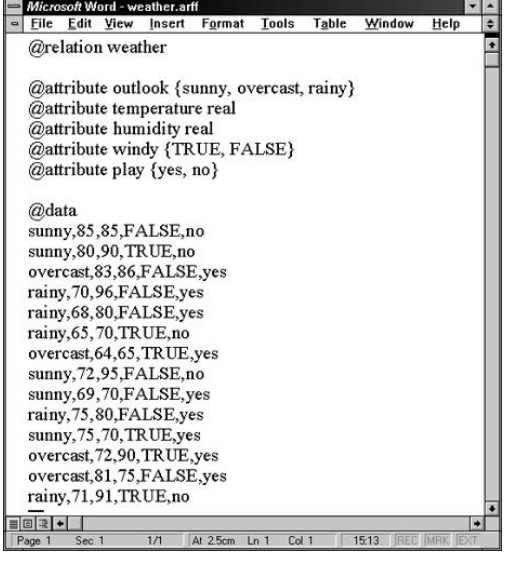


	A	B	C	D	E
1	outlook	temperatu	humidity	windy	play
2					
3	sunny	85	85	FALSE	no
4	sunny	80	90	TRUE	no
5	overcast	83	86	FALSE	yes
6	rainy	70	96	FALSE	yes
7	rainy	68	80	FALSE	yes
8	rainy	65	70	TRUE	no
9	overcast	64	65	TRUE	yes
10	sunny	72	95	FALSE	no
11	sunny	69	70	FALSE	yes
12	rainy	75	80	FALSE	yes
13	sunny	75	70	TRUE	yes
14	overcast	72	90	TRUE	yes
15	overcast	81	75	FALSE	yes
16	rainy	71	91	TRUE	no
17					
18					
19					
20					

(a)



(b)



(c)

Figure 8.1 Weather data: (a) in spreadsheet; (b) comma-separated; (c) in ARFF format.

(Comma Delimited) from the file type popup menu, enter a name for the file, and click the *Save* button. (A message will warn you that this will only save the active sheet: just ignore it by clicking *OK*.)

Now load this file into Microsoft Word. Your screen will look like Figure 8.1b. The rows of the original spreadsheet have been converted into lines of text, and the elements are separated from each other by commas. All you have to do is convert the first line, which holds the attribute names, into the header structure that makes up the beginning of an ARFF file.

Figure 8.1c shows the result. The dataset's name is introduced by a `@relation` tag, and the names, types, and values of each attribute are defined by `@attribute` tags. The data section of the ARFF file begins with a `@data` tag. Once the structure of your dataset matches Figure 8.1c, you should save it as a text file. Choose *Save as...* from the *File* menu, and specify *Text Only with Line Breaks* as the file type by using the corresponding popup menu. Enter a file name, and press the *Save* button. We suggest that you rename the file to *weather.arff* to indicate that it is in ARFF format. Note that the classification schemes in Weka assume by default that the class is the last attribute in the ARFF file, which fortunately it is in this case. (We explain in Section 8.3 below how to override this default.)

Now you can start analyzing this data using the algorithms provided. In the following we assume that you have downloaded Weka to your system, and that your Java environment knows where to find the library. (More information on how to do this can be found at the Weka Web site.)

To see what the C4.5 decision tree learner described in Section 6.1 does with this dataset, we use the J4.8 algorithm, which is Weka's implementation of this decision tree learner. (J4.8 actually implements a later and slightly improved version called C4.5 Revision 8, which was the last public version of this family of algorithms before C5.0, a commercial implementation, was released.) Type

```
java weka.classifiers.j48.J48 -t weather.arff
```

at the command line. This incantation calls the Java virtual machine and instructs it to execute the `J48` algorithm from the *j48* package—a subpackage of *classifiers*, which is part of the overall *weka* package. Weka is organized in "packages" that correspond to a directory hierarchy. We'll give more details of the package structure in the next section: in this case, the subpackage name is *j48* and the program to be executed from it is called `J48`. The `-t` option informs the algorithm that the next argument is the name of the training file.

After pressing *Return*, you'll see the output shown in Figure 8.2. The first part is a pruned decision tree in textual form. As you can see, the first split is on the `outlook` attribute, and then, at the second level, the splits are on `humidity` and `windy`, respectively. In the tree structure, a colon introduces the class label that has been assigned to a particular leaf, followed by the number of instances that reach that leaf, expressed as a

```
J48 pruned tree
------------------

outlook = sunny
|   humidity <= 75: yes (2.0)
|   humidity > 75: no (3.0)
outlook = overcast: yes (4.0)
outlook = rainy
|   windy = TRUE: no (2.0)
|   windy = FALSE: yes (3.0)

Number of Leaves  :      5
Size of the tree :      8


=== Error on training data ===

Correctly Classified Instances          14               100      %
Incorrectly Classified Instances         0                 0      %
Mean absolute error                      0
Root mean squared error                  0
Total Number of Instances               14

=== Confusion Matrix ===

 a b   <-- classified as
 9 0 | a = yes
 0 5 | b = no

=== Stratified cross-validation ===

Correctly Classified Instances           9               64.2857 %
Incorrectly Classified Instances         5               35.7143 %
Mean absolute error                      0.3036
Root mean squared error                  0.4813
Total Number of Instances               14

=== Confusion Matrix ===

 a b   <-- classified as
 7 2 | a = yes
 3 2 | b = no
```

Figure 8.2 Output from the J4.8 decision tree learner.

decimal number because of the way the algorithm uses fractional instances to handle missing values. Below the tree structure, the number of leaves is printed, then the total number of nodes in the tree (`Size of the tree`).

The second part of the output gives estimates of the tree's predictive performance, generated by Weka's evaluation module. The first set of measurements is derived from the training data. As discussed in Section 5.1, such measurements are highly optimistic and very likely to overestimate the true predictive performance. However, it is still useful to look at these results, for they generally represent an upper bound on the model's performance on fresh data. In this case, all fourteen training instances have been classified correctly, and none were left unclassified. An instance can be left unclassified if the learning scheme refrains from assigning any class label to it, in which case the number of unclassified instances will be reported in the output. For most learning schemes in Weka, this never occurs.

In addition to the classification error, the evaluation module also outputs measurements derived from the class probabilities assigned by the tree. More specifically, it outputs the mean absolute error and the root mean-squared error of the probability estimates. The root mean-squared error is the square root of the average quadratic loss, discussed in Section 5.6. The mean absolute error is calculated in a similar way by using the absolute instead of the squared difference. In this example, both figures are 0 because the output probabilities for the tree are either 0 or 1, due to the fact that all leaves are pure and all training instances are classified correctly.

The summary of the results from the training data ends with a confusion matrix, mentioned in Chapter 5 (Section 5.7), showing how many instances of each class have been assigned to each class. In this case, only the diagonal elements of the matrix are non-zero because all instances are classified correctly.

The final section of the output presents results obtained using stratified ten-fold cross-validation. The evaluation module automatically performs a ten-fold cross-validation if no test file is given. As you can see, more than 30% of the instances (5 out of 14) have been misclassified in the cross-validation. This indicates that the results obtained from the training data are very optimistic compared with what might be obtained from an independent test set from the same source. From the confusion matrix you can observe that two instances of class `yes` have been assigned to class `no`, and three of class `no` are assigned to class `yes`.

8.2 Javadoc and the class library

Before exploring other learning algorithms, it is useful to learn more about

the structure of Weka. The most detailed and up-to-date information can be found in the online documentation on the Weka Web site. This documentation is generated directly from comments in the source code using Sun's Javadoc utility. To understand its structure, you need to know how Java programs are organized.

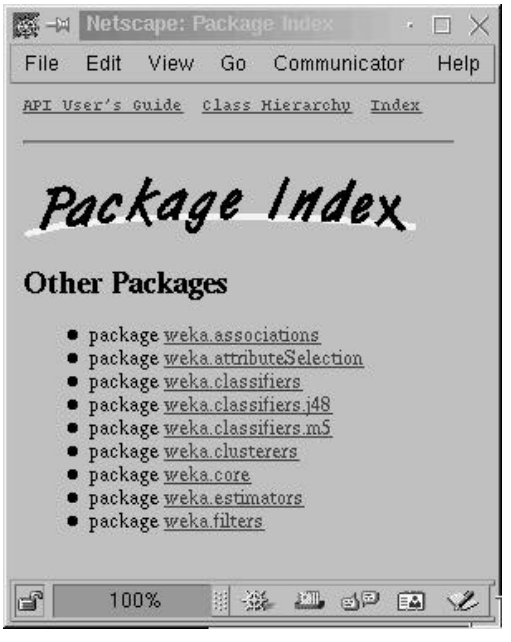
Classes, instances, and packages

Every Java program is implemented as a *class.* In object-oriented programming, a class is a collection of variables along with some *methods* that operate on those variables. Together, they define the behavior of an *object* belonging to the class. An object is simply an instantiation of the class that has values assigned to all the class's variables. In Java, an object is also called an *instance* of the class. Unfortunately this conflicts with the terminology used so far in this book, where the terms *class* and *instance* have appeared in the quite different context of machine learning. From now on, you will have to infer the intended meaning of these terms from the context in which they appear. This is not difficult—though sometimes we'll use the word *object* instead of Java's *instance* to make things clear.

In Weka, the implementation of a particular learning algorithm is represented by a class. We have already met one, the `J48` class described above that builds a C4.5 decision tree. Each time the Java virtual machine executes `J48`, it creates an instance of this class by allocating memory for building and storing a decision tree classifier. The algorithm, the classifier it builds, and a procedure for outputting the classifier, are all part of that instantiation of the `J48` class.

Larger programs are usually split into more than one class. The `J48` class, for example, does not actually contain any code for building a decision tree. It includes references to instances of other classes that do most of the work. When there are a lot of classes—as in Weka—they can become difficult to comprehend and navigate. Java allows classes to be organized into *packages.* A package is simply a directory containing a collection of related classes. The *j48* package mentioned above contains the classes that implement J4.8, our version of C4.5, and PART, which is the name we use for the scheme for building rules from partial decision trees that was explained near the end of Section 6.2 (page 181). Not surprisingly, these two learning algorithms share a lot of functionality, and most of the classes in this package are used by both algorithms, so it is logical to put them in the same place. Because each package corresponds to a directory, packages are organized in a hierarchy. As already mentioned, the *j48* package is a subpackage of the *classifiers* package, which is itself a subpackage of the overall *weka* package.

When you consult the online documentation generated by Javadoc from your Web browser, the first thing you see is a list of all the packages in



(a)

(b)

Figure 8.3 Using Javadoc: (a) the front page; (b) the *weka.core* package.

Weka (Figure 8.3a). In the following we discuss what each one contains. On the Web page they are listed in alphabetical order; here we introduce them in order of importance.

The weka.core package

The *core* package is central to the Weka system. It contains classes that are accessed from almost every other class. You can find out what they are by clicking on the hyperlink underlying *weka.core*, which brings up Figure 8.3b.

The Web page in Figure 8.3b is divided into two parts: the *Interface Index* and the *Class Index*. The latter is a list of all classes contained within the package, while the former lists all the interfaces it provides. An *interface* is very similar to a class, the only difference being that it doesn't actually do anything by itself—it is merely a list of methods without actual implementations. Other classes can declare that they "implement" a particular interface, and then provide code for its methods. For example, the `OptionHandler` interface defines those methods that are implemented by all classes that can process command-line options—including all classifiers.

The key classes in the *core* package are called `Attribute`, `Instance`, and `Instances`. An object of class `Attribute` represents an attribute. It contains the attribute's name, its type and, in the case of a nominal attribute, its possible values. An object of class `Instance` contains the attribute values of a particular instance; and an object of class `Instances` holds an ordered set of instances, in other words, a dataset. By clicking on the hyperlinks underlying the classes, you can find out more about them. However, you need not know the details just to use Weka from the command line. We will return to these classes in Section 8.4 when we discuss how to access the machine learning routines from other Java code.

Clicking on the *All Packages* hyperlink in the upper left corner of any documentation page brings you back to the listing of all the packages in Weka (Figure 8.3a).

The weka.classifiers package

The *classifiers* package contains implementations of most of the algorithms for classification and numeric prediction that have been discussed in this book. (Numeric prediction is included in *classifiers:* it is interpreted as prediction of a continuous class.) The most important class in this package is `Classifier`, which defines the general structure of any scheme for classification or numeric prediction. It contains two methods, `buildClassifier()` and `classifyInstance()`, which all of these learning algorithms have to implement. In the jargon of object-oriented programming, the learning algorithms are represented by subclasses of `Classifier`, and therefore automatically inherit these two methods. Every scheme redefines them according to how it builds a classifier and how it classifies instances. This gives a uniform interface for building and using classifiers from other Java code. Hence, for example, the same evaluation module can be used to evaluate the performance of any classifier in Weka.

Another important class is `DistributionClassifier`. This subclass of `Classifier` defines the method `distributionForInstance()`, which returns a probability distribution for a given instance. Any classifier that can calculate class probabilities is a subclass of `DistributionClassifier` and implements this method.

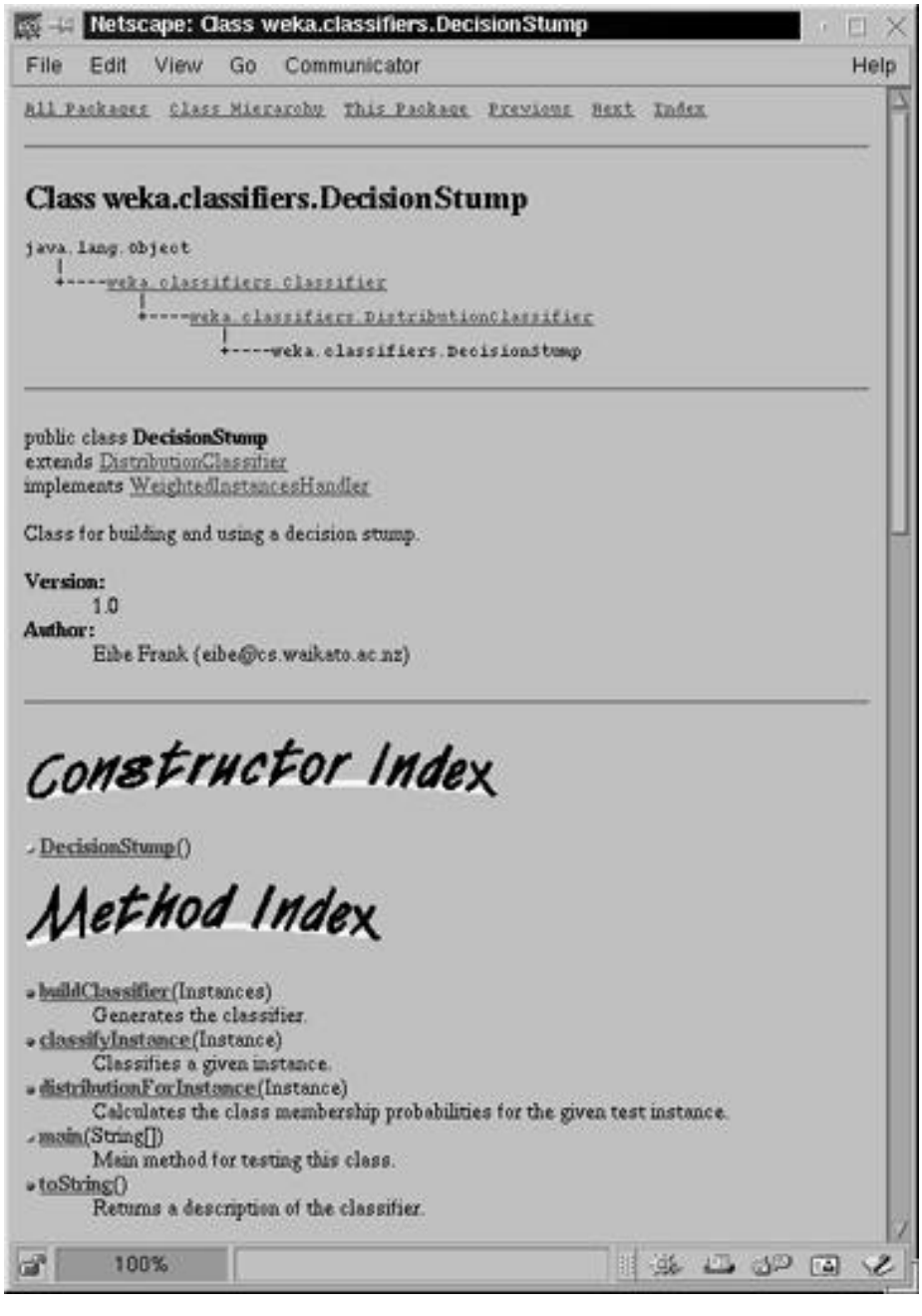


Figure 8.4 A class of the *weka.classifiers* package.

To see an example, click on `DecisionStump`, which is a class for building a simple one-level binary decision tree (with an extra branch for missing values). Its documentation page, shown in Figure 8.4, begins with the fully qualified name of this class: `weka.classifiers.DecisionStump`. You have to use this rather lengthy expression if you want to build a decision stump from the command line. The page then displays a tree structure showing the relevant part of the class hierarchy. As you can see, `DecisionStump` is a subclass of `DistributionClassifier`, and therefore produces class probabilities. `DistributionClassifier`, in turn, is a subclass of `Classifier`, which is itself a subclass of `Object`. The `Object` class is the most general one in Java: all classes are automatically subclasses of it.

After some generic information about the class, its author, and its version, Figure 8.4 gives an index of the constructors and methods of this class. A *constructor* is a special kind of method that is called whenever an object of that class is created, usually initializing the variables that collectively define its state. The index of methods lists the name of each one, the type of parameters it takes, and a short description of its functionality. Beneath those indexes, the Web page gives more details about the constructors and methods. We return to those details later.

As you can see, `DecisionStump` implements all methods required by both a `Classifier` and a `DistributionClassifier`. In addition, it contains `toString()` and `main()` methods. The former returns a textual description of the classifier, used whenever it is printed on the screen. The latter is called every time you ask for a decision stump from the command line, in other words, every time you enter a command beginning with

```
java weka.classifiers.DecisionStump
```

The presence of a `main()` method in a class indicates that it can be run from the command line, and all learning methods and filter algorithms implement it.

Other packages

Several other packages listed in Figure 8.3a are worth mentioning here: *weka.classifiers.j48*, *weka.classifiers.m5*, *weka.associations*, *weka.clusterers*, *weka.estimators*, *weka.filters*, and *weka.attributeSelection*. The *weka.classifiers.j48* package contains the classes implementing J4.8 and the PART rule learner. They have been placed in a separate package (and hence in a separate directory) to avoid bloating the *classifiers* package. The *weka.classifiers.m5* package contains classes implementing the model tree algorithm of Section 6.5, which is called M5′.

In Chapter 4 (Section 4.5) we discussed an algorithm for mining association rules, called APRIORI. The *weka.associations* package contains two classes, `ItemSet` and `Apriori`, which together implement this algorithm.

They have been placed in a separate package because association rules are fundamentally different from classifiers. The *weka.clusterers* package contains an implementation of two methods for unsupervised learning: COBWEB and the EM algorithm (Section 6.6). The *weka.estimators* package contains subclasses of a generic `Estimator` class, which computes different types of probability distribution. These subclasses are used by the Naive Bayes algorithm.

Along with actual learning schemes, tools for preprocessing a dataset, which we call *filters*, are an important component of Weka. In *weka.filters*, the `Filter` class is the analog of the `Classifier` class described above. It defines the general structure of all classes containing filter algorithms—they are all implemented as subclasses of `Filter`. Like classifiers, filters can be used from the command line; we will see later how this is done. It is easy to identify classes that implement filter algorithms: their names end in *Filter*.

Attribute selection is an important technique for reducing the dimensionality of a dataset. The *weka.attributeSelection* package contains several classes for doing this. These classes are used by the `AttributeSelectionFilter` from *weka.filters*, but they can also be used separately.

Indexes

As mentioned above, all classes are automatically subclasses of `Object`. This makes it possible to construct a tree corresponding to the hierarchy of all classes in Weka. You can examine this tree by selecting the *Class Hierarchy* hyperlink from the top of any page of the online documentation. This shows very concisely which classes are subclasses or superclasses of a particular class—for example, which classes inherit from `Classifier`.

The online documentation contains an index of all publicly accessible variables (called *fields*) and methods in Weka—in other words, all fields and methods that you can access from your own Java code. To view it, click on the *Index* hyperlink located at the top of every documentation page.

8.3 Processing datasets using the machine learning programs

We have seen how to use the online documentation to find out which learning methods and other tools are provided in the Weka system. Now we show how to use these algorithms from the command line, and then discuss them in more detail.

```
Pruned training model tree:

MMAX <= 14000 : LM1 (141/4.18%)
MMAX >  14000 : LM2 (68/51.8%)

Models at the leaves (smoothed):

  LM1:  class = 4.15
                - 2.05vendor=honeywell,ipl,ibm,cdc,ncr,basf,
                     gould,siemens,nas,adviser,sperry,amdahl
                + 5.43vendor=adviser,sperry,amdahl
                - 5.78vendor=amdahl + 0.00638MYCT
                + 0.00158MMIN + 0.00345MMAX
                + 0.552CACH + 1.14CHMIN + 0.0945CHMAX
  LM2:  class = -113
                - 56.1vendor=honeywell,ipl,ibm,cdc,ncr,basf,
                     gould,siemens,nas,adviser,sperry,amdahl
                + 10.2vendor=adviser,sperry,amdahl
                - 10.9vendor=amdahl
                + 0.012MYCT + 0.0145MMIN + 0.0089MMAX
                + 0.808CACH + 1.29CHMAX

=== Error on training data ===

Correlation coefficient                  0.9853
Mean absolute error                     13.4072
Root mean squared error                 26.3977
Relative absolute error                 15.3431 %
Root relative squared error             17.0985 %
Total Number of Instances              209

=== Cross-validation ===

Correlation coefficient                  0.9767
Mean absolute error                     13.1239
Root mean squared error                 33.4455
Relative absolute error                 14.9884 %
Root relative squared error             21.6147 %
Total Number of Instances              209
```

Figure 8.5 Output from the M5′ program for numeric prediction.

Using M5´

Section 8.1 explained how to interpret the output of a decision tree learner and showed the performance figures that are automatically generated by the evaluation module. The interpretation of these is the same for all models that predict a categorical class. However, when evaluating models for numeric prediction, Weka produces a different set of performance measures.

As an example, suppose you have a copy of the CPU performance dataset from Table 1.5 of Chapter 1 named *cpu.arff* in the current directory. Figure 8.5 shows the output obtained if you run the model tree inducer M5Ì on it by typing

```
java weka.classifiers.m5.M5Prime -t cpu.arff
```

and pressing *Return*. The structure of the pruned model tree is surprisingly simple. It is a decision stump, a binary 1-level decision tree, with a split on the `MMAX` attribute. Attached to that stump are two linear models, one for each leaf. Both involve one nominal attribute, called `vendor`. The expression `vendor=adviser, sperry,amdahl` is interpreted as follows: if `vendor` is either `adviser`, `sperry`, or `amdahl`, then substitute 1, otherwise 0.

The description of the model tree is followed by several figures that measure its performance. As with decision tree output, the first set is derived from the training data and the second uses tenfold cross-validation (this time not stratified, of course, because that doesn't make sense for numeric prediction). The meaning of the different measures is explained in Section 5.8.

Generic options

In the examples above, the `-t` option was used to communicate the name of the training file to the learning algorithm. There are several other options that can be used with any learning scheme, and also scheme-specific ones that apply only to particular schemes. If you invoke a scheme without any command-line options at all, it displays all options that can be used. First the general options are listed, then the scheme-specific ones. Try, for example,

```
java weka.classifiers.j48.J48
```

You'll see a listing of the options common to all learning schemes, shown in Table 8.1, followed by a list of those that just apply to J48, shown in Table 8.2. We will explain the generic options and then briefly review the scheme-specific ones.

Table 8.1 Generic options for learning schemes in Weka.

option	function
`-t <training file>`	Specify training file
`-T <test file>`	Specify test file. If none, a cross-validation is performed on the training data
`-c <class index>`	Specify index of class attribute
`-x <number of folds>`	Specify number of folds for cross-validation
`-s <random number seed>`	Specify random number seed for cross-validation
`-m <cost matrix file>`	Specify file containing cost matrix
`-v`	Output no statistics for training data
`-l <input file>`	Specify input file for model
`-d <output file>`	Specify output file for model
`-o`	Output statistics only, not the classifier
`-I`	Output information retrieval statistics for two-class problems
`-k`	Output information-theoretic statistics
`-p`	Only output predictions for test instances
`-r`	Only output cumulative margin distribution

The options in Table 8.1 determine which data is used for training and testing, how the classifier is evaluated, and what kind of statistics are displayed. You might want to use an independent test set instead of performing a cross-validation on the training data to evaluate a learning scheme. The `-T` option allows just that: if you provide the name of a file, the data in it is used to derive performance statistics, instead of cross-validation. Sometimes the class is not the last attribute in an ARFF file: you can declare that another one is the class using `-c`. This option requires you to specify the position of the desired attribute in the file, 1 for the first attribute, 2 for the second, and so on. When tenfold cross-validation is performed (the default if a test file is not provided), the data is randomly shuffled first. If you want to repeat the cross-validation several times, each time reshuffling the data in a different way, you can set the random number seed with `-s` (default value 1). With a large dataset you may want to reduce the number of folds for the cross-validation from the default value of 10 using `-x`.

Weka also implements cost-sensitive classification. If you provide the name of a file containing a cost matrix using the `-m` option, the dataset will be reweighted (or resampled, depending on the learning scheme) according to the information in this file. Here is a cost matrix for the weather data above:

```
0 1 10 % If true class yes and prediction no, penalty is 10
1 0 1  % If true class no and prediction yes, penalty is 1
```

Each line must contain three numbers: the index of the true class, the index of the incorrectly assigned class, and the penalty, which is the amount by which that particular error will be weighted (the penalty must be a positive number). Not all combinations of actual and predicted classes need be listed: the default penalty is 1. (In all Weka input files, comments introduced by % can be appended to the end of any line.)

```
J48 pruned tree
------------------
: yes (14.0/0.74)

Number of Rules  :      1
Size of the tree :      1

=== Confusion Matrix ===

 a b   <-- classified as
 9 0 | a = yes
 5 0 | b = no

=== Stratified cross-validation ===

Correctly Classified Instances         9               64.2857 %
Incorrectly Classified Instances       5               35.7143 %
Correctly Classified With Cost        90               94.7368 %
Incorrectly Classified With Cost       5                5.2632 %
Mean absolute error                    0.3751
Root mean squared error                0.5714
Total Number of Instances             14
Total Number With Cost                95

=== Confusion Matrix ===

 a b   <-- classified as
 9 0 | a = yes
 5 0 | b = no
```

Figure 8.6 Output from J4.8 with cost-sensitive classification.

To illustrate cost-sensitive classification, let's apply J4.8 to the weather data, with a heavy penalty if the learning scheme predicts `no` when the true class is `yes`. Save the cost matrix above in a file called *costs* in the same directory as *weather.arff*. Assuming that you want the cross-validation performance only, not the error on the training data, enter

```
java weka.classifiers.j48.J48 -t weather.arff -m costs -v
```

The output, shown in Figure 8.6, is quite different from that given earlier in Figure 8.2. To begin with, the decision tree has been reduced to its root! Also, four new performance measures are included, each one ending in `With Cost`. These are calculated by weighting the instances according to the weights given in the cost matrix. As you can see, the learner has decided that it's best to always predict `yes` in this situation—which is not surprising, given the heavy penalty for erroneously predicting `no`.

Returning to Table 8.1, it is also possible to save and load models. If you provide the name of an output file using `-d`, Weka will save the classifier generated from the training data into this file. If you want to evaluate the same classifier on a new batch of test instances, you can load it back using `-l` instead of rebuilding it. If the classifier can be updated incrementally (and you can determine this by checking whether it implements the `UpdateableClassifier` interface), you can provide both a training file and an input file, and Weka will load the classifier and update it with the given training instances.

If you only wish to assess the performance of a learning scheme and are not interested in the model itself, use `-o` to suppress output of the model. To see the information-retrieval performance measures of precision, recall, and the F-measure that were introduced in Section 5.7, use `-i` (note that these can only be calculated for two-class datasets). Information-theoretic measures computed from the probabilities derived by a learning

Table 8.2 Scheme-specific options for the J4.8 decision tree learner.

option	function
`-U`	Use unpruned tree
`-C <pruning confidence>`	Specify confidence threshold for pruning
`-M <number of instances>`	Specify minimum number of instances in a leaf
`-R`	Use reduced-error pruning
`-N <number of folds>`	Specify number of folds for reduced error pruning. One fold is used as pruning set
`-S`	Use binary splits only
	Don't perform subtree raising

scheme—such as the informational loss function discussed in Section 5.6—can be obtained with `-k`.

Users often want to know which class values the learning scheme actually predicts for each test instance. The `-p` option, which only applies if you provide a test file, prints the number of each test instance, its class, the confidence of the scheme's prediction, and the predicted class value. Finally, you can output the cumulative margin distribution for the training data. This allows you to investigate how the distribution of the margin measure from Section 7.4 (in the subsection *Boosting*) changes with the number of iterations performed when boosting a learning scheme.

Scheme-specific options

Table 8.2 shows the options specific to J4.8. You can force the algorithm to use the unpruned tree instead of the pruned one. You can suppress subtree raising, which results in a more efficient algorithm. You can set the confidence threshold for the pruning procedure, and the minimum number of instances permissible at any leaf—both parameters were discussed in Section 6.1 (p. 169). In addition to C4.5's standard pruning procedure, reduced-error pruning (Section 6.2) can be performed, which prunes the decision tree to optimize performance on a holdout set. The `-N` option governs how large this set is: the dataset is divided equally into that number of parts, and the last is used as the holdout set (default value 3). Finally, to build a binary tree instead of one with multiway branches for nominal attributes, use `-B`.

Classifiers

J4.8 is just one of many practical learning schemes that you can apply to your dataset. Table 8.3 lists them all, giving the name of the class implementing the scheme along with its most important scheme-specific options and their effects. It also indicates whether the scheme can handle weighted instances (W column), whether it can output a class distribution for datasets with a categorical class (D column), and whether it can be updated incrementally (I column). Table 8.3 omits a few other schemes designed mainly for pedagogical purposes that implement some of the basic methods covered in Chapter 4—a rudimentary implementation of Naive Bayes, a divide-and-conquer decision tree algorithm (ID3), a covering algorithm for generating rules (PRISM), and a nearest-neighbor instance-based learner (IB1); we will say something about these in Section 8.5 when we explain how to write new machine learning schemes. Of course, Weka is a growing system: other learning algorithms will be added in due course, and the online documentation must be consulted for a definitive list.

Table 8.3 The learning schemes in Weka.

scheme	book section	class	W	D	I	option	function
Majority/average predictor		`weka.classifiers.ZeroR`	y	y	n	None	
1R	4.1	`weka.classifiers.OneR`	n	n	n	`-B <>`	Specify minimum bucket size
Naive Bayes	4.2	`weka.classifiers.NaiveBayes`	y	y	n	`-K`	Use kernel density estimator
Decision table	3.1	`weka.classifiers.DecisionTable`	y	y	n	`-X <>`	Specify number of folds for cross-validation
						`-S <>`	Specify threshold for stopping search
						`-I`	Use nearest-neighbor classifier
Instance-based learner	4.7	`weka.classifiers.IBk`	y	y	y	`-D`	Weight by inverse of distance
						`-F`	Weight by 1-distance
						`-K <>`	Specify number of neighbors
						`-W <>`	Specify window size
						`-X`	Use cross-validation
C4.5	6.1	`weka.classifiers.j48.J48`	y	y	n	Table 8.2	Already discussed
PART rule learner	6.2	`weka.classifiers.j48.PART`	y	y	n	Table 8.2	As for J4.8, except that *–U* and *–S* are not available
Support vector machine	6.3	`weka.classifiers.SMO`	n	y	n	`-C <>`	Specify upper bound for weights
						`-E <>`	Specify degree of polynomials
Linear regression	4.6	`weka.classifiers.LinearRegression`	y	–	n	`-S <>`	Specify attribute selection method
M5' model tree learner	6.5	`weka.classifiers.m5.M5Prime`	n	–	n	`-O <>`	Specify type of model
						`-U`	Use unsmoothed tree
						`-F <>`	Specify pruning factor
						`-V <>`	Specify verbosity of output
Locally weighted regression	6.5	`weka.classifiers.LWR`	y	–	y	`-K <>`	Specify number of neighbors
						`-W <>`	Specify kernel shape
One-level decision trees	7.4	`weka.classifiers.DecisionStump`	y	y	n	None	

The most primitive of the schemes in Table 8.3 is called `ZeroR`: it simply predicts the majority class in the training data if the class is categorical and the average class value if it is numeric. Although it makes little sense to use this scheme for prediction, it can be useful for determining a baseline performance as a benchmark for other learning schemes. (Sometimes other schemes actually perform worse than `ZeroR`: this indicates serious overfitting.)

Ascending the complexity ladder, the next learning scheme is `OneR`, discussed in Section 4.1, which produces simple rules based on one attribute only. It takes a single parameter: the minimum number of instances that must be covered by each rule that is generated (default value 6).

`NaiveBayes` implements the probabilistic Naive Bayesian classifier from Section 4.2. By default it uses the normal distribution to model numeric attributes; however, the `-K` option instructs it to use kernel density estimators instead. This can improve performance if the normality assumption is grossly incorrect.

The next scheme in Table 8.3, `DecisionTable`, produces a decision table using the wrapper method of Section 7.1 to find a good subset of attributes for inclusion in the table. This is done using a best-first search. The number of non-improving attribute subsets that are investigated before the search terminates can be controlled using `-S` (default value 5). The number of cross-validation folds performed by the wrapper can be changed using `-X` (default: leave-one-out). Usually, a decision table assigns the majority class from the training data to a test instance if it does not match any entry in the table. However, if you specify the `-I` option, the nearest match will be used instead. This often improves performance significantly.

`IBk` is an implementation of the *k*-nearest-neighbors classifier that employs the distance metric discussed in Section 4.7. By default it uses just one nearest neighbor ($k = 1$), but the number can be specified manually with `-K` or determined automatically using leave-one-out cross-validation. The `-X` option instructs `IBk` to use cross-validation to determine the best value of *k* between 1 and the number given by `-K`. If more than one neighbor is selected, the predictions of the neighbors can be weighted according to their distance to the test instance, and two different formulas are implemented for deriving the weight from the distance (`-D` and `-F`). The time taken to classify a test instance with a nearest-neighbor classifier increases linearly with the number of training instances. Consequently it is sometimes necessary to restrict the number of training instances that are kept in the classifier, which is done by setting the window size option.

We have already discussed the options for J4.8; those for PART, which forms rules from pruned partial decision trees built using C4.5's heuristics as described near the end of Section 6.2 (page 181), are a subset of these.

Just as reduced-error pruning can reduce the size of a J4.8 decision tree, it can also reduce the number of rules produced by PART—with the side effect of decreasing run time because complexity depends on the number of rules that are generated. However, reduced-error pruning often reduces the accuracy of the resulting decision trees and rules because it reduces the amount of data that can be used for training. With large enough datasets, this disadvantage vanishes.

In Section 6.3 we introduced support vector machines. The `SMO` class implements the *sequential minimal optimization* algorithm, which learns this type of classifier. Despite being one of the fastest methods for learning support vector machines, sequential minimal optimization is often slow to converge to a solution—particularly when the data is not linearly separable in the space spanned by the nonlinear mapping. Because of noise, this often happens. Both run time and accuracy depend critically on the values that are given to two parameters: the upper bound on the coefficients' values in the equation for the hyperplane (`-C`), and the degree of the polynomials in the non-linear mapping (`-E`). Both are set to 1 by default. The best settings for a particular dataset can be found only by experimentation.

The next three learning schemes in Table 8.3 are for numeric prediction. The simplest is linear regression, whose only parameter controls how attributes to be included in the linear function are selected. By default, the heuristic employed by the model tree inducer M5′ is used, whose run time is linear in the number of attributes. However, it is possible to suppress all attribute selection by setting `-S` to 1, and to use greedy forward selection, whose run time is quadratic in the number of attributes, by setting `-S` to 2.

The class that implements M5′ has already been described in the example on page 279. It implements the algorithm explained in Section 6.5 except that a simpler method is used to deal with missing values: they are replaced by the global mean or mode of the training data before the model tree is built. Several different forms of model output are provided, controlled by the `-O` option: a model tree (`-O m`), a regression tree without linear models at the leaves (`-O r`), and a simple linear regression (`-O l`). The automatic smoothing procedure described in Section 6.5 can be disabled using `-U`. The amount of pruning that this algorithm performs can be controlled by setting the pruning factor to a value between 0 and 10. Finally, the verbosity of the output can be set to a value from 0 to 3.

Locally weighted regression, the second scheme for numeric prediction described in Section 6.5, is implemented by the `LWR` class. Its performance depends critically on the correct choice of kernel width, which is determined by calculating the distance of the test instance to its *k*th nearest neighbor. The value of *k* can be specified using `-K`. Another factor that influences performance is the shape of the kernel: choices are 0 for a

Table 8.4 The meta-learning schemes in Weka.

scheme	option	function
weka.classifiers.Bagging	-I <>	Specify number of iterations
	-W <>	Specify base learner
	-S <>	Specify random number seed
weka.classifiers.AdaBoostM1	-I <>	Specify number of iterations
	-P <>	Specify weight mass to be used
	-W <>	Specify base learner
	-Q	Use resampling
	-S <>	Specify random number seed
weka.classifiers.LogitBoost	-I <>	Specify number of iterations
	-P <>	Specify weight mass to be used
	-W <>	Specify base learner
weka.classifiers. MultiClassClassifier	-W <>	Specify base learner
weka.classifiers. CVParameterSelection	-W <>	Specify base learner
	-P <>	Specify option to be optimized
	-X <>	Specify number of cross-validation folds
	-S <>	Specify random number seed
weka.classifiers. Stacking	-B <>	Specify level-0 learner and options
	-M <>	Specify level-1 learner and options
	-X <>	Specify number of cross-validation folds
	-S <>	Specify random number seed

linear kernel (the default), 1 for an inverse one, and 2 for the classic Gaussian kernel.

The final scheme in Table 8.3, `DecisionStump`, builds binary decision stumps—one-level decision trees—for datasets with either a categorical or a numeric class. It copes with missing values by extending a third branch from the stump, in other words, by treating `missing` as a separate attribute value. It is designed for use with the boosting methods discussed later in this section.

Meta-learning schemes

Chapter 7 described methods for enhancing the performance and extending the capabilities of learning schemes. We call these *meta-learning schemes* because they incorporate other learners. Like ordinary learning

schemes, meta learners belong to the *classifiers* package: they are summarized in Table 8.4.

The first is an implementation of the bagging procedure discussed in Section 7.4. You can specify the number of bagging iterations to be performed (default value 10), and the random number seed for resampling. The name of the learning scheme to be bagged is declared using the `-W` option. Here is the beginning of a command line for bagging unpruned J4.8 decision trees:

```
java weka.classifiers.bagging -W jaws.classifiers.j48.J48...-- -U
```

There are two lists of options, those intended for bagging and those for the base learner itself, and a double minus sign (`--`) is used to separate the lists. Thus the `-U` in the above command line is directed to the `J48` program, where it will cause the use of unpruned trees (see Table 8.2). This convention avoids the problem of conflict between option letters for the meta learner and those for the base learner.

`AdaBoost.M1`, also discussed in Section 7.4, is handled in the same way as bagging. However, there are two additional options. First, if `-Q` is used, boosting with resampling will be performed instead of boosting with reweighting. Second, the `-P` option can be used to accelerate the learning process: in each iteration only the percentage of the weight mass specified by `-P` is passed to the base learner, instances being sorted according to their weight. This means that the base learner has to process fewer instances because often most of the weight is concentrated on a fairly small subset, and experience shows that the consequent reduction in classification accuracy is usually negligible.

Another boosting procedure is implemented by `LogitBoost`. A detailed discussion of this method is beyond the scope of this book; suffice to say that it is based on the concept of additive logistic regression (Friedman et al. 1998). In contrast to `AdaBoost.M1`, `LogitBoost` can successfully boost very simple learning schemes, (like `DecisionStump`, that was introduced above), even in multiclass situations. From a user's point of view, it differs from `AdaBoost.M1` in an important way because it boosts schemes for numeric prediction in order to form a combined classifier that predicts a categorical class.

Weka also includes an implementation of a meta learner which performs stacking, as explained in Chapter 7 (Section 7.4). In stacking, the result of a set of different level-0 learners is combined by a level-1 learner. Each level-0 learner must be specified using `-B`, followed by any relevant options—and the entire specification of the level-0 learner, including the options, must be enclosed in double quotes. The level-1 learner is specified in the same way, using `-M`. Here is an example:

```
java weka.classifiers.Stacking -B "weka.classifiers.j48.J48 -U"
```

```
-B "weka.classifiers.IBk -K 5" -M "weka.classifiers.j48.J48" ...
```

By default, tenfold cross-validation is used; this can be changed with the `-X` option.

Some learning schemes can only be used in two-class situations—for example, the `SMO` class described above. To apply such schemes to multiclass datasets, the problem must be transformed into several two-class ones and the results combined. `MultiClassClassifier` does exactly that: it takes a base learner that can output a class distribution or a numeric class, and applies it to a multiclass learning problem using the simple one-per-class coding introduced in Section 4.6 (p. 114).

Often, the best performance on a particular dataset can only be achieved by tedious parameter tuning. Weka includes a meta learner that performs optimization automatically using cross-validation. The `-W` option of `CVParameterSelection` takes the name of a base learner, and the `-P` option specifies one parameter in the format

```
"<option name> <starting value> <last value> <number of steps>"
```

An example is:

```
java...CVParameterSelection -W...OneR -P "B 1 10 10" -t
   weather.arff
```

which evaluates integer values between 1 and 10 for the `B` parameter of 1R. Multiple parameters can be specified using several `-P` options.

`CVParameterSelection` causes the space of all possible combinations of the given parameters to be searched exhaustively. The parameter set with the best cross-validation performance is chosen, and this is used to build a classifier from the full training set. The `-X` option allows you to specify the number of folds (default 10).

Suppose you are unsure of the capabilities of a particular classifier—for example, you might want to know whether it can handle weighted

```
@relation weather-weka.filters.AttributeFilter-R1_2

@attribute humidity real
@attribute windy {TRUE,FALSE}
@attribute play {yes,no}

@data

85,FALSE,no
90,TRUE,no
...
```

Figure 8.7 Effect of `AttributeFilter` on the weather dataset.

Table 8.5 The filter algorithms in Weka.

filter	option	function
weka.filters.AddFilter	-C <>	Specify index of new attribute
	-L <>	Specify labels for nominal attribute
	-N <>	Specify name of new attribute
weka.filters.	-E <>	Specify evaluation class
AttributeSelectionFilter	-S <>	Specify search class
	-T <>	Set threshold by which to discard attributes
weka.filters.AttributeFilter	-R <>	Specify attributes to be deleted
	-V	Invert matching sense
weka.filters.DiscretizeFilter	-B <>	Specify number of bins
	-O	Optimize number of bins
	-R <>	Specify attributes to be discretized
	-V	Invert matching sense
	-D	Output binary attributes
weka.filter.MakeIndicatorFilter	-C <>	Specify attribute index
	-V <>	Specify value index
	-N	Output nominal attribute
weka.filter.MergeTwoValuesFilter	-C <>	Specify attribute index
	-F <>	Specify first value index
	-S <>	Specify second value index

instances. The `weka.classifiers.CheckClassifier` tool prints a summary of any classifier's properties. For example,

```
java weka.classifiers.CheckClassifier -W weka.classifiers.IBk
```

prints a summary of the properties of the `IBk` class discussed above.

In Section 7.4 we discussed the bias-variance decomposition of a learning algorithm. Weka includes an algorithm that estimates the bias and variance of a particular learning scheme with respect to a given dataset. `BVDecompose` takes the name of a learning scheme and a training file and performs a bias-variance decomposition. It provides options for setting the index of the class attribute, the number of iterations to be performed, and the random number seed. The more iterations that are performed, the better the estimate.

Table 8.5 The filter algorithms in Weka. (continued)

filter	option	function
weka.filters. NominalToBinaryFilter	-N	Output nominal attributes
weka.filters. ReplaceMissingValuesFilter		
weka.filters.InstanceFilter	-C <>	Specify attribute index
	-S <>	Specify numeric value
	-L <>	Specify nominal values
	-V	Invert matching sense
weka.filters. SwapAttributeValuesFilter	-C <>	Specify attribute index
	-F <>	Specify first value index
	-S <>	Specify second value index
weka.filters. NumericTransformFilter	-R <>	Specify attributes to be transformed
	-V	Invert matching sense
	-C <>	Specify Java class
	-M <>	Specify transformation method
weka.filters. SplitDatasetFilter	-R <>	Specify range of instances to be split
	-V	Invert matching sense
	-N <>	Specify number of folds
	-F <>	Specify fold to be returned
	-S <>	Specify random number seed

Filters

Having discussed the learning schemes in the *classifiers* package, we now turn to the next important package for command-line use, *filters*. We begin by examining a simple filter that can be used to delete specified attributes from a dataset, in other words, to perform manual attribute selection. The following command line

```
java weka.filters.AttributeFilter -R 1,2 -i weather.arff
```

yields the output in Figure 8.7. As you can see, attributes 1 and 2, namely `outlook` and `temperature`, have been deleted. Note that no spaces are allowed in the list of attribute indices. The resulting dataset can be placed in the file *weather.new.arff* by typing:

```
java...AttributeFilter -R 1,2 -i weather.arff -o weather.new.arff
```

All filters in Weka are used in the same way. They take an input file

specified using the `-i` option and an optional output file specified with `-o`. A class index can be specified using `-c`. Filters read the ARFF file, transform it, and write it out. (If files are not specified, they read from standard input and write to standard output, so that they can be used as a "pipe" in Unix systems.) All filter algorithms provide a list of available options in response to `-h`, as in

```
java weka.filters.AttributeFilter -h
```

Table 8.5 lists the filters implemented in Weka, along with their principal options.

The first, `AddFilter`, inserts an attribute at the given position. For all instances in the dataset, the new attribute's value is declared to be missing. If a list of comma-separated nominal values is given using the `-L` option, the new attribute will be a nominal one, otherwise it will be numeric. The attribute's name can be set with `-N`.

`AttributeSelectionFilter` allows you to select a set of attributes using different methods: since it is rather complex we will leave it to last.

`AttributeFilter` has already been used above. However, there is a further option: if `-V` is used the matching set is inverted—that is, only attributes *not* included in the `-R` specification are deleted.

An important filter for practical applications is `DiscretizeFilter`. It contains an unsupervised and a supervised discretization method, both discussed in Section 7.2. The former implements equal-width binning, and the number of bins can be set manually using `-B`. However, if `-O` is present, the number of bins will be chosen automatically using a cross-validation procedure that maximizes the estimated likelihood of the data. In that case, `-B` gives an upper bound to the possible number of bins. If the index of a class attribute is specified using `-c`, supervised discretization will be performed using the MDL method of Fayyad and Irani (1993). Usually, discretization loses the ordering implied by the original numeric attribute when it is transformed into a nominal one. However, this information is preserved if the discretized attribute with k values is transformed into k-1 binary attributes. The `-D` option does this automatically by producing one binary attribute for each split point (described in Section 7.2 [p. 239]).

`MakeIndicatorFilter` is used to convert a nominal attribute into a binary indicator attribute and can be used to transform a multiclass dataset into several two-class ones. The filter substitutes a binary attribute for the chosen nominal one, setting the corresponding value for each instance to 1 if a particular original value was present and to 0 otherwise. Both the attribute to be transformed and the original nominal value are set by the user. By default the new attribute is declared to be numeric, but if `-N` is given it will be nominal.

Suppose you want to merge two values of a nominal attribute into a

single category. This is done by `MergeAttributeValuesFilter`. The name of the new value is a concatenation of the two original ones, and every occurrence of either of the original values is replaced by the new one. The index of the new value is the smaller of the original indexes.

Some learning schemes, like support vector machines, can handle only binary attributes. The advantage of binary attributes is that they can be treated as either nominal or numeric. `NominalToBinaryFilter` transforms all multivalued nominal attributes in a dataset into binary ones, replacing each attribute with k values by $k-1$ binary attributes. If a class is specified using the `-c` option, it is left untouched. The transformation used for the other attributes depends on whether the class is numeric. If the class is numeric, the M5′ transformation method is employed for each attribute; otherwise a simple one-per-value encoding is used. If the `-N` option is used, all new attributes will be nominal, otherwise they will be numeric.

One way of dealing with missing values is to replace them globally before applying a learning scheme. `ReplaceMissingValuesFilter` substitutes the mean, for numeric attributes, or the mode, for nominal ones, for each occurrence of a missing value.

To remove from a dataset all instances that have certain values for nominal attributes, or numeric values above or below a certain threshold, use `InstanceFilter`. By default all instances are deleted that exhibit one of a given set of nominal attribute values (if the specified attribute is nominal), or a numeric value below a given threshold (if it is numeric). However, the matching criterion can be inverted using `-V`.

The `SwapAttributeValuesFilter` is a simple one: all it does is swap the positions of two values of a nominal attribute. Of course, this could also be accomplished by editing the ARFF file in a word processor. The order of attribute values is entirely cosmetic: it does not affect machine learning at all. If the selected attribute is the class, changing the order affects the layout of the confusion matrix.

In some applications it is necessary to transform a numeric attribute before a learning scheme is applied—for example, replacing each value with its square root. This is done using `NumericTransformFilter`, which transforms all of the selected numeric attributes using a given function. The transformation can be any Java function that takes a `double` as its argument and returns another `double`, for example, `sqrt()` in `java.lang.Math`. The name of the class that implements the function (which must be a fully qualified name) is set using `-C`, and the name of the transformation method is set using `-M`: thus to take the square root use:

```
java weka.filters.NumericTransformFilter -C java.lang.Math -M sqrt...
```

Weka also includes a filter with which you can generate subsets of a dataset, `SplitDatasetFilter`. You can either supply a range of instances to be selected using the `-R` option, or generate a random subsample whose

size is determined by the `-N` option. The dataset is split into the given number of folds, and one of them (indicated by `-F`) is returned. If a random number seed is provided (with `-S`), the dataset will be shuffled before the subset is extracted. Moreover, if a class attribute is set using `-c` the dataset will be stratified, so that the class distribution in the subsample is approximately the same as in the full dataset.

It is often necessary to apply a filter algorithm to a training dataset and then, using settings derived from the training data, apply the same filter to a test file. Consider a filter that discretizes numeric attributes. If the discretization method is supervised—that is, if it uses the class values to derive good intervals for the discretization—the results will be biased if it is applied directly to the test data. It is the discretization intervals derived from the *training* data that must be applied to the test data. More generally, the filter algorithm must optimize its internal settings according to the training data and apply these same settings to the test data. This can be done in a uniform way with all filters by adding `-b` as a command-line option and providing the name of input and output files for the test data using `-r` and `-s` respectively. Then the filter class will derive its internal settings using the training data provided by `-i` and use these settings to transform the test data.

Finally, we return to `AttributeSelectionFilter`. This lets you select a set of attributes using attribute selection classes in the `weka.attributeSelection` package. The `-c` option sets the class index for supervised attribute selection. With `-E` you provide the name of an evaluation class from `weka.attributeSelection` that determines how the filter evaluates attributes, or sets of attributes; in addition you may need to use `-S` to specify a search technique. Each feature evaluator, subset evaluator, and search method has its own options. They can be printed with `-h`.

There are two types of evaluators that you can specify with `-E`: ones that consider one attribute at a time, and ones that consider sets of attributes together. The former are subclasses of `weka.attributeSelection.AttributeEvaluator`—an example is `weka.attributeSelection.InfoGainAttributeEval`, which evaluates attributes according to their information gain. The latter are subclasses of `weka.attributeSelection.SubsetEvaluator`—like `weka.attributeSelection.CfsSubsetEval`, which evaluates subsets of features by the correlation among them. If you give the name of a subclass of `AttributeEvaluator`, you must also provide, using `-T`, a threshold by which the filter can discard low-scoring attributes. On the other hand, if you give the name of a subclass of `SubsetEvaluator`, you must provide the name of a search class using `-S`, which is used to search through possible subsets of attributes. Any subclass of `weka.attributeSelection.ASSearch` can be used for this option—for

```
Apriori
=======

Minimum support: 0.2
Minimum confidence: 0.9
Number of cycles performed: 17

Generated sets of large itemsets:

Size of set of large itemsets L(1): 12
Size of set of large itemsets L(2): 47
Size of set of large itemsets L(3): 39
Size of set of large itemsets L(4): 6

Best rules found:

1 . humidity=normal windy=FALSE 4 ==> play=yes 4 (1)
2 . temperature=cool 4 ==> humidity=normal 4 (1)
3 . outlook=overcast 4 ==> play=yes 4 (1)
4 . temperature=cool play=yes 3 ==> humidity=normal 3 (1)
5 . outlook=rainy windy=FALSE 3 ==> play=yes 3 (1)
6 . outlook=rainy play=yes 3 ==> windy=FALSE 3 (1)
7 . outlook=sunny humidity=high 3 ==> play=no 3 (1)
8 . outlook=sunny play=no 3 ==> humidity=high 3 (1)
9 . temperature=cool windy=FALSE 2 ==> humidity=normal play=yes 2 (1)
10. temperature=cool humidity=normal windy=FALSE 2 ==> play=yes 2 (1)
```

Figure 8.8 Output from the APRIORI association rule learner.

example `weka.attributeSelection.BestFirst`, which implements a best-first search.

Here is an example showing `AttributeSelectionFilter` being used with correlation-based subset evaluation and best-first search for the weather data:

```
java weka.filters.AttributeSelectionFilter
   -S weka.attributeSelection.BestFirst
   -E weka.attributeSelection.CfsSubsetEval
   -i weather.arff -c5
```

To provide options for the evaluator, you must enclose both the name of the evaluator and its options in double quotes (e.g., `-S "<evaluator> <options>"`). Options for the search class can be specified in the same way.

Table 8.6 Principal options for the APRIORI association rule learner.

option	function
`-t <training file>`	Specify training file
`-N <required number of rules>`	Specify required number of rules
`-C <minimum confidence of a rule>`	Specify minimum confidence of a rule
`-D <delta for minimum support>`	Specify delta for decrease of minimum support
`-M <lower bound for minimum support>`	Specify lower bound for minimum support

Association rules

Weka includes an implementation of the APRIORI algorithm for generating association rules: the class for this is `weka.associations.Apriori`. To see what it does, try

```
java weka.associations.Apriori –t weather.nominal.arff
```

where `weather.nominal.arff` is the nominal version of the weather data from Section 1.2. (The APRIORI algorithm can only deal with nominal attributes.)

The output is shown in Figure 8.8. The last part gives the association rules that are found. The number preceding the `==>` symbol indicates the rule's support, that is, the number of items covered by its premise. Following the rule is the number of those items for which the rule's consequent holds as well. In parentheses is the confidence of the rule—in other words, the second figure divided by the first. In this simple example, the confidence is 1 for every rule. APRIORI orders rules according to their confidence and uses support as a tiebreaker. Preceding the rules are the numbers of item sets found for each support size considered. In this case six item sets of four items were found to have the required minimum

Table 8.7 Generic options for clustering schemes in Weka.

option	function
`-t <training file>`	Specify training file
`-T <test file>`	Specify test file
`-x <number of folds>`	Specify number of folds for cross-validation
`-s <random number seed>`	Specify random number seed for cross-validation
`-l <input file>`	Specify input file for model
`-d <output file>`	Specify output file for model
`-p`	Only output predictions for test instances

support.

By default, APRIORI tries to generate ten rules. It begins with a minimum support of 100% of the data items and decreases this in steps of 5% until there are at least ten rules with the required minimum confidence, or until the support has reached a lower bound of 10%, whichever occurs first. The minimum confidence is set to 0.9 by default. As you can see from the beginning of Figure 8.8, the minimum support decreased to 0.2, or 20%, before the required number of rules could be generated; this involved a total of 17 iterations.

All of these parameters can be changed by setting the corresponding options. As with other learning algorithms, if the program is invoked without any command-line arguments, all applicable options are listed. The principal ones are summarized in Table 8.6.

Clustering

Weka includes a package that contains clustering algorithms, *weka.clusterers*. These operate in a similar way to the classification methods in *weka.classifiers*. The command-line options are again split into generic and scheme-specific options. The generic ones, summarized in Table 8.7, are just the same as for classifiers except that a cross-validation is not performed by default if the test file is missing.

It may seem strange that there is an option for providing test data. However, if clustering is accomplished by modeling the distribution of instances probabilistically, it is possible to check how well the model fits the data by computing the likelihood of a set of test data given the model. Weka measures goodness-of-fit by the logarithm of the likelihood, or log-likelihood: and the larger this quantity, the better the model fits the data. Instead of using a single test set, it is also possible to compute a cross-validation estimate of the log-likelihood using `-x`.

Weka also outputs how many instances are assigned to each cluster. For clustering algorithms that do not model the instance distribution probabilistically, these are the only statistics that Weka outputs. It's easy to find out which clusterers generate a probability distribution: they are subclasses of `weka.clusterers.DistributionClusterer`.

There are two clustering algorithms in *weka.clusterers:* `weka.clusterers.EM` and `weka.clusterers.Cobweb`. The former is an implementation of the `EM` algorithm and the latter implements the incremental clustering algorithm (both are described in Chapter 6, Section 6.6). They can handle both numeric and nominal attributes.

Like Naive Bayes, `EM` makes the assumption that the attributes are independent random variables. The command line

```
java weka.clusterers.EM -t weather.arff -N 2
```

results in the output shown in Figure 8.9. The `-N` options forces `EM` to generate two clusters. As you can see, the number of clusters is printed first, followed by a description of each one: the cluster's prior probability and a probability distribution for all attributes in the dataset. For a nominal attribute, the distribution is represented by the count associated with each value (plus one); for a numeric attribute it is a standard normal distribution. `EM` also outputs the number of training instances in each cluster, and the log-likelihood of the training data with respect to the clustering that it generates.

By default, `EM` selects the number of clusters automatically by maximizing the logarithm of the likelihood of future data, estimated using cross-validation. Beginning with one cluster, it continues to add clusters until the estimated log-likelihood decreases. However, if you have access to prior knowledge about the number of clusters in your data, it makes sense to force `EM` to generate the desired number of clusters. Apart from `-N`, `EM` recognizes two additional scheme-specific command-line options: `-I` sets the maximum number of iterations performed by the algorithm, and `-S` sets the random number seed used to initialize the cluster membership probabilities.

The cluster hierarchy generated by COBWEB is controlled by two parameters: the acuity and the cutoff (see Chapter 6, page 216). They can be set using the command-line options `-A` and `-C`, and are given as a percentage. COBWEB's output is very sensitive to these parameters, and it pays to spend some time experimenting with them.

8.4 Embedded machine learning

When invoking learning schemes and filter algorithms from the command line, there is no need to know anything about programming in Java. In this section we show how to access these algorithms from your own code. In doing so, the advantages of using an object-oriented programming language will become clear. From now on, we assume that you have at least some rudimentary knowledge of Java. In most practical applications of data mining, the learning component is an integrated part of a far larger software environment. If the environment is written in Java, you can use Weka to solve the learning problem without writing any machine learning code yourself.

A simple message classifier

We present a simple data mining application, automatic classification of email messages, to illustrate how to access classifiers and filters. Because its purpose is educational, the system has been kept as simple as possible, and

```
EM
==

Number of clusters: 2

Cluster: 0 Prior probability: 0.2816

Attribute: outlook
Discrete Estimator. Counts =  2.96 2.98 1  (Total = 6.94)
Attribute: temperature
Normal Distribution. Mean =  82.2692 StdDev =   2.2416
Attribute: humidity
Normal Distribution. Mean =  83.9788 StdDev =   6.3642
Attribute: windy
Discrete Estimator. Counts =  1.96 3.98  (Total = 5.94)
Attribute: play
Discrete Estimator. Counts =  2.98 2.96  (Total = 5.94)

Cluster: 1 Prior probability: 0.7184

Attribute: outlook
Discrete Estimator. Counts =  4.04 3.02 6  (Total = 13.06)
Attribute: temperature
Normal Distribution. Mean =  70.1616 StdDev =   3.8093
Attribute: humidity
Normal Distribution. Mean =  80.7271 StdDev =  11.6349
Attribute: windy
Discrete Estimator. Counts =  6.04 6.02  (Total = 12.06)
Attribute: play
Discrete Estimator. Counts =  8.02 4.04  (Total = 12.06)


=== Clustering stats for training data ===

Cluster Instances
      0         4  (29 %)
      1        10  (71 %)

Log likelihood: -9.01881
```

Figure 8.9 Output from the EM clustering scheme.

it certainly doesn't perform at the state of the art. However, it will give you an idea how to use Weka in your own application. Furthermore, it is straightforward to extend the system to make it more sophisticated.

The first problem faced when trying to apply machine learning in a practical setting is selecting attributes for the data at hand. This is probably also the most important problem: if you don't choose meaningful attributes—attributes which together convey sufficient information to make learning tractable—any attempt to apply machine learning techniques is doomed to fail. In truth, the choice of a learning scheme is usually far less important than coming up with a suitable set of attributes.

In the example application, we do not aspire to optimum performance, so we use rather simplistic attributes: they count the number of times specific keywords appear in the message to be classified. We assume that each message is stored in an individual file, and the program is called every time a new message is to be processed. If the user provides a class label for the message, the system will use the message for training; if not, it will try to classify it. The instance-based classifier `IBk` is used for this example application.

Figure 8.10 shows the source code for the application program, implemented in a class called `MessageClassifier`. The `main()` method accepts the following command-line arguments: the name of a message file (given by `-m`), the name of a file holding an object of class `MessageClassifier` (`-t`) and, optionally, the classification of the message (`-c`). The message's class can be `hit` or `miss`. If the user provides a classification using `-c`, the message will be added to the training data; if not, the program will classify the message as either `hit` or `miss`.

Main()

The `main()` method reads the message into an array of characters and checks whether the user has provided a classification for it. It then attempts to read an existing `MessageClassifier` object from the file given by `-t`. If this file does not exist, a new object of class `MessageClassifier` will be created. In either case the resulting object is called `messageCl`. After checking for illegal command-line options, the given message is used to either update `messageCl` by calling the method `updateModel()` on it, or classify it by calling `classifyMessage()`. Finally, if `messageCl` has been updated, the object is saved back into the file. In the following, we first discuss how a new `MessageClassifier` object is created by the constructor `MessageClassifier()`, and then explain how the two methods `updateModel()` and `classifyMessage()` work.

MessageClassifier()

Each time a new `MessageClassifier` is created, objects for holding a dataset, a filter, and a classifier are generated automatically. The only nontrivial

```
/**
* Java program for classifying short text messages into two classes.
 */

import weka.core.*;
import weka.classifiers.*;
import weka.filters.*;
import java.io.*;
import java.util.*;

public class MessageClassifier implements Serializable {

  /* Our (rather arbitrary) set of keywords. */
  private final String[] m_Keywords = {"product", "only", "offer", "great",
   "amazing", "phantastic", "opportunity", "buy", "now"};

  /* The training data. */
  private Instances m_Data = null;

  /* The filter. */
  private Filter m_Filter = new DiscretizeFilter();

  /* The classifier. */
  private Classifier m_Classifier = new IBk();

  /**
   * Constructs empty training dataset.
   */
  public MessageClassifier() throws Exception {

    String nameOfDataset = "MessageClassificationProblem";

    // Create numeric attributes.
    FastVector attributes = new FastVector(m_Keywords.length + 1);
    for (int i = 0 ; i < m_Keywords.length; i++) {
      attributes.addElement(new Attribute(m_Keywords[i]));
    }

    // Add class attribute.
    FastVector classValues = new FastVector(2);
    classValues.addElement("miss");
    classValues.addElement("hit");
    attributes.addElement(new Attribute("Class", classValues));

    // Create dataset with initial capacity of 100, and set index of class.
    m_Data = new Instances(nameOfDataset, attributes, 100);
    m_Data.setClassIndex(m_Data.numAttributes() - 1);
  }

  /**
   * Updates model using the given training message.
   */
  public void updateModel(String message, String classValue)
    throws Exception {
```

Figure 8.10 Source code for the message classifier.

```
    // Convert message string into instance.
    Instance instance = makeInstance(cleanupString(message));

    // Add class value to instance.
    instance.setClassValue(classValue);

    // Add instance to training data.
    m_Data.add(instance);

    // Use filter.
    m_Filter.inputFormat(m_Data);
    Instances filteredData = Filter.useFilter(m_Data, m_Filter);

    // Rebuild classifier.
    m_Classifier.buildClassifier(filteredData);
  }

  /**
   * Classifies a given message.
   */
  public void classifyMessage(String message) throws Exception {

    // Check if classifier has been built.
    if (m_Data.numInstances() == 0) {
      throw new Exception("No classifier available.");
    }

    // Convert message string into instance.
    Instance instance = makeInstance(cleanupString(message));

    // Filter instance.
    m_Filter.input(instance);
    Instance filteredInstance = m_Filter.output();

    // Get index of predicted class value.
    double predicted = m_Classifier.classifyInstance(filteredInstance);

    // Classify instance.
    System.err.println("Message classified as : " +
                   m_Data.classAttribute().value((int)predicted));
  }

  /**
   * Method that converts a text message into an instance.
   */
  private Instance makeInstance(String messageText) {


    StringTokenizer tokenizer = new StringTokenizer(messageText);
    Instance instance = new Instance(m_Keywords.length + 1);
    String token;

    // Initialize counts to zero.
    for (int i = 0; i < m_Keywords.length; i++) {
```

Figure 8.10 (continued)

```
      instance.setValue(i, 0);
    }

    // Compute attribute values.
    while (tokenizer.hasMoreTokens()) {
      token = tokenizer.nextToken();
      for (int i = 0; i < m_Keywords.length; i++) {
        if (token.equals(m_Keywords[i])) {
          instance.setValue(i, instance.value(i) + 1.0);
          break;
        }
      }
    }

    // Give instance access to attribute information from the dataset.
    instance.setDataset(m_Data);

    return instance;
  }

  /**
   * Method that deletes all non-letters from a string, and lowercases it.
   */
  private String cleanupString(String messageText) {

    char[] result = new char[messageText.length()];
    int position = 0;

    for (int i = 0; i < messageText.length(); i++) {
      if (Character.isLetter(messageText.charAt(i)) ||
        Character.isWhitespace(messageText.charAt(i))) {
        result[position++] = Character.toLowerCase(messageText.charAt(i));
      }
    }
    return new String(result);
  }

  /**
   * Main method.
   */
  public static void main(String[] options) {

    MessageClassifier messageCl;
    byte[] charArray;

    try {

      // Read message file into string.
      String messageFileString = Utils.getOption('m', options);
      if (messageFileString.length() != 0) {
        FileInputStream messageFile = new FileInputStream(messageFileString);
        int numChars = messageFile.available();
```

Figure 8.10 (continued)

```
        charArray = new byte[numChars];
        messageFile.read(charArray);
        messageFile.close();
      } else {
        throw new Exception ("Name of message file not provided.");
      }

      // Check if class value is given.
      String classValue = Utils.getOption('c', options);

      // Check for model file. If existent, read it, otherwise create new
      // one.
      String modelFileString = Utils.getOption('t', options);
      if (modelFileString.length() != 0) {
        try {
          FileInputStream modelInFile = new FileInputStream(modelFileString);
          ObjectInputStream modelInObjectFile =
            new ObjectInputStream(modelInFile);
          messageCl = (MessageClassifier) modelInObjectFile.readObject();
          modelInFile.close();
        } catch (FileNotFoundException e) {
          messageCl = new MessageClassifier();
        }
      } else {
        throw new Exception ("Name of data file not provided.");
      }

      // Check if there are any options left
      Utils.checkForRemainingOptions(options);

      // Process message.
      if (classValue.length() != 0) {
        messageCl.updateModel(new String(charArray), classValue);
      } else {
        messageCl.classifyMessage(new String(charArray));
      }

      // If class has been given, updated message classifier must be saved
      if (classValue.length() != 0) {
        FileOutputStream modelOutFile =
          new FileOutputStream(modelFileString);
        ObjectOutputStream modelOutObjectFile =
          new ObjectOutputStream(modelOutFile);
        modelOutObjectFile.writeObject(messageCl);
        modelOutObjectFile.flush();
        modelOutFile.close();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Figure 8.10

part of the process is creating a dataset, which is done by the constructor `MessageClassifier()`. First the dataset's name is stored as a string. Then an `Attribute` object is created for each of the attributes—one for each keyword, and one for the class. These objects are stored in a dynamic array of type `FastVector`. (`FastVector` is Weka's own fast implementation of the standard Java `Vector` class. `Vector` is implemented in a way that allows parallel programs to synchronize access to them, which was very slow in early Java implementations.)

Attributes are created by invoking one of the two constructors in the class `Attribute`. The first takes one parameter—the attribute's name—and creates a numeric attribute. The second takes two parameters: the attribute's name and a `FastVector` holding the names of its values. This latter constructor generates a nominal attribute. In `MessageClassifier`, the attributes for the keywords are numeric, so only their names need be passed to `Attribute()`. The keyword itself is used to name the attribute. Only the class attribute is nominal, with two values: `hit` and `miss`. Hence, `MessageClassifier()` passes its name ("`class`") and the values—stored in a `FastVector`—to `Attribute()`.

Finally, to create a dataset from this attribute information, `MessageClassifier()` must create an object of the class `Instances` from the *core* package. The constructor of `Instances` used by `MessageClassifier()` takes three arguments: the dataset's name, a `FastVector` containing the attributes, and an integer indicating the dataset's initial capacity. We set the initial capacity to 100; it is expanded automatically if more instances are added to the dataset. After constructing the dataset, `MessageClassifier()` sets the index of the class attribute to be the index of the last attribute.

UpdateModel()

Now that you know how to create an empty dataset, consider how the `MessageClassifier` object actually incorporates a new training message. The method `updateModel()` does this job. It first calls `cleanupString()` to delete all nonletters and non-whitespace characters from the message. Then it converts the message into a training instance by calling `makeInstance()`. The latter method counts the number of times each of the keywords in `m_Keywords` appears in the message, and stores the result in an object of the class `Instance` from the *core* package. The constructor of `Instance` used in `makeInstance()` sets all the instance's values to be missing, and its weight to 1. Therefore `makeInstance()` must set all attribute values other than the class to 0 before it starts to calculate keyword frequencies.

Once the message has been processed, `makeInstance()` gives the newly created instance access to the data's attribute information by passing it a reference to the dataset. In Weka, an `Instance` object does not store the type of each attribute explicitly; instead it stores a reference to a dataset with the corresponding attribute information.

Returning to `updateModel()`, once the new instance has been returned from `makeInstance()` its class value is set and it is added to the training data. In the next step a filter is applied to this data. In our application the `DiscretizeFilter` is used to discretize all numeric attributes. Because a class index has been set for the dataset, the filter automatically uses supervised discretization (otherwise equal-width discretization would be used). Before the data can be transformed, we must first inform the filter of its format. This is done by passing it a reference to the corresponding input dataset via `inputFormat()`. Every time this method is called, the filter is initialized—that is, all its internal settings are reset. In the next step, the data is transformed by `useFilter()`. This generic method from the `Filter` class applies a given filter to a given dataset. In this case, because the `DiscretizeFilter` has just been initialized, it first computes discretization intervals from the training dataset, then uses these intervals to discretize it. After returning from `useFilter()`, all the filter's internal settings are fixed until it is initialized by another call of `inputFormat()`. This makes it possible to filter a test instance without updating the filter's internal settings.

In the last step, `updateModel()` rebuilds the classifier—in our program, an instance-based `IBk` classifier—by passing the training data to its `buildClassifier()` method. It is a convention in Weka that the `buildClassifier()` method completely initializes the model's internal settings before generating a new classifier.

ClassifyMessage()

Now we consider how `MessageClassifier` processes a test message—a message for which the class label is unknown. In `classifyMessage()`, our program first checks that a classifier has been constructed by seeing if any training instances are available. It then uses the methods described above—`cleanupString()` and `makeInstance()`—to transform the message into a test instance. Because the classifier has been built from filtered training data, the test instance must also be processed by the filter before it can be classified. This is very easy: the `input()` method enters the instance into the filter object, and the transformed instance is obtained by calling `output()`. Then a prediction is produced by passing the instance to the classifier's `classifyInstance()` method. As you can see, the prediction is coded as a `double` value. This allows Weka's evaluation module to treat models for categorical and numeric prediction similarly. In the case of categorical prediction, as in this example, the `double` variable holds the index of the predicted class value. In order to output the string corresponding to this class value, the program calls the `value()` method of the dataset's class attribute.

8.5 Writing new learning schemes

Suppose you need to implement a special-purpose learning algorithm that is not included in Weka, or a filter that performs an unusual data transformation. Or suppose you are engaged in machine learning research and want to investigate a new learning scheme or data preprocessing operation. Or suppose you just want to learn more about the inner workings of an induction algorithm by actually programming it yourself. This section shows how to make full use of Weka's class hierarchy when writing classifiers and filters, using a simple example of each.

Several elementary learning schemes, not mentioned above, are included in Weka mainly for educational purposes: they are listed in Table 8.8. None of them takes any scheme-specific command-line options. All these implementations are useful for understanding the inner workings of a classifier. As an example, we discuss the `weka.classifiers.Id3` scheme, which implements the ID3 decision tree learner from Section 4.3.

An example classifier

Figure 8.11 gives the source code of `weka.classifiers.Id3`, which, as you can see from the code, extends the `DistributionClassifier` class. This means that in addition to the `buildClassifier()` and `classifyInstance()` methods from the `Classifier` class it also implements the `distributionForInstance()` method, which returns a predicted distribution of class probabilities for an instance. We will study the implementation of these three methods in turn.

BuildClassifier()

The `buildClassifier()` method constructs a classifier from a set of training data. In our implementation it first checks the training data for a non-nominal class, missing values, or any other attribute that is not nominal, because the ID3 algorithm can't handle these. It then makes a copy of the training set (to avoid changing the original data) and calls a method from

Table 8.8 Simple learning schemes in Weka.

scheme	description	section
`weka.classifiers.NaiveBayesSimple`	Probabilistic learner	4.2
`weka.classifiers.Id3`	Decision tree learner	4.3
`weka.classifiers.Prism`	Rule learner from	4.4
`weka.classifiers.IB1`	Instance-based learner	4.7

```
import weka.classifiers.*;
import weka.core.*;
import java.io.*;
import java.util.*;

/**
 * Class implementing an Id3 decision tree classifier.
 */
public class Id3 extends DistributionClassifier {

  /** The node's successors. */
  private Id3[] m_Successors;

  /** Attribute used for splitting. */
  private Attribute m_Attribute;

  /** Class value if node is leaf. */
  private double m_ClassValue;

  /** Class distribution if node is leaf. */
  private double[] m_Distribution;

  /** Class attribute of dataset. */
  private Attribute m_ClassAttribute;

  /**
   * Builds Id3 decision tree classifier.
   */
  public void buildClassifier(Instances data) throws Exception {

    if (!data.classAttribute().isNominal()) {
      throw new Exception("Id3: nominal class, please.");
    }
    Enumeration enumAtt = data.enumerateAttributes();
    while (enumAtt.hasMoreElements()) {
      Attribute attr = (Attribute) enumAtt.nextElement();
      if (!attr.isNominal()) {
        throw new Exception("Id3: only nominal attributes, please.");
      }
      Enumeration enum = data.enumerateInstances();
      while (enum.hasMoreElements()) {
        if (((Instance) enum.nextElement()).isMissing(attr)) {
          throw new Exception("Id3: no missing values, please.");
        }
      }
    }
    data = new Instances(data);
    data.deleteWithMissingClass();
    makeTree(data);
  }

  /**
   * Method building Id3 tree.
   */
  private void makeTree(Instances data) throws Exception {

    // Check if no instances have reached this node.
```

Figure 8.11 Source code for the ID3 decision tree learner.

```
    if (data.numInstances() == 0) {
      m_Attribute = null;
      m_ClassValue = Instance.missingValue();
      m_Distribution = new double[data.numClasses()];
      return;
    }

    // Compute attribute with maximum information gain.
    double[] infoGains = new double[data.numAttributes()];
    Enumeration attEnum = data.enumerateAttributes();
    while (attEnum.hasMoreElements()) {
      Attribute att = (Attribute) attEnum.nextElement();
      infoGains[att.index()] = computeInfoGain(data, att);
    }
    m_Attribute = data.attribute(Utils.maxIndex(infoGains));

    // Make leaf if information gain is zero.
    // Otherwise create successors.
    if (Utils.eq(infoGains[m_Attribute.index()], 0)) {
      m_Attribute = null;
      m_Distribution = new double[data.numClasses()];
      Enumeration instEnum = data.enumerateInstances();
      while (instEnum.hasMoreElements()) {
        Instance inst = (Instance) instEnum.nextElement();
        m_Distribution[(int) inst.classValue()]++;
        }
      Utils.normalize(m_Distribution);
      m_ClassValue = Utils.maxIndex(m_Distribution);
      m_ClassAttribute = data.classAttribute();
    } else {
      Instances[] splitData = splitData(data, m_Attribute);
      m_Successors = new Id3[m_Attribute.numValues()];
      for (int j = 0; j < m_Attribute.numValues(); j++) {
        m_Successors[j] = new Id3();
        m_Successors[j].buildClassifier(splitData[j]);
      }
    }
  }

  /**
   * Classifies a given test instance using the decision tree.
   */

  public double classifyInstance(Instance instance) {

    if (m_Attribute == null) {
      return m_ClassValue;
    } else {
      return m_Successors[(int) instance.value(m_Attribute)].
          classifyInstance(instance);
    }
  }

  /**
   * Computes class distribution for instance using decision tree.
   */
  public double[] distributionForInstance(Instance instance) {
```

Figure 8.11 (continued)

```
    if (m_Attribute == null) {
      return m_Distribution;
    } else {
      return m_Successors[(int) instance.value(m_Attribute)].
          distributionForInstance(instance);
    }
  }

  /**
   * Prints the decision tree using the private toString method from below.
   */
  public String toString() {

    return "Id3 classifier\n==============\n" + toString(0);
  }

  /**
   * Computes information gain for an attribute.
   */
  private double computeInfoGain(Instances data, Attribute att)
    throws Exception {

    double infoGain = computeEntropy(data);
    Instances[] splitData = splitData(data, att);
    for (int j = 0; j < att.numValues(); j++) {
      if (splitData[j].numInstances() > 0) {
        infoGain -= ((double) splitData[j].numInstances() /
                     (double) data.numInstances()) *
          computeEntropy(splitData[j]);
      }
    }
    return infoGain;
  }

 /**
  * Computes the entropy of a dataset.
  */
  private double computeEntropy(Instances data) throws Exception {

    double [] classCounts = new double[data.numClasses()];
    Enumeration instEnum = data.enumerateInstances();
    while (instEnum.hasMoreElements()) {
      Instance inst = (Instance) instEnum.nextElement();
      classCounts[(int) inst.classValue()]++;
    }
    double entropy = 0;
    for (int j = 0; j < data.numClasses(); j++) {
      if (classCounts[j] > 0) {
        entropy -= classCounts[j] * Utils.log2(classCounts[j]);
      }
    }
    entropy /= (double) data.numInstances();
    return entropy + Utils.log2(data.numInstances());
  }

  /**
   * Splits a dataset according to the values of a nominal attribute.
```

Figure 8.11 (continued)

```
   */
  private Instances[] splitData(Instances data, Attribute att) {

    Instances[] splitData = new Instances[att.numValues()];
    for (int j = 0; j < att.numValues(); j++) {
      splitData[j] = new Instances(data, data.numInstances());
    }
    Enumeration instEnum = data.enumerateInstances();
    while (instEnum.hasMoreElements()) {
      Instance inst = (Instance) instEnum.nextElement();
      splitData[(int) inst.value(att)].add(inst);
    }
    return splitData;
  }

  /**
   * Outputs a tree at a certain level.
   */
  private String toString(int level) {

    StringBuffer text = new StringBuffer();

    if (m_Attribute == null) {
      if (Instance.isMissingValue(m_ClassValue)) {
        text.append(": null");

      } else {
        text.append(": "+m_ClassAttribute.value((int) m_ClassValue));
      }
   } else {
      for (int j = 0; j < m_Attribute.numValues(); j++) {
        text.append("\n");
        for (int i = 0; i < level; i++) {
          text.append("|  ");
        }
        text.append(m_Attribute.name() + " = " + m_Attribute.value(j));
        text.append(m_Successors[j].toString(level + 1));
      }
    }
    return text.toString();
  }

  /**
   * Main method.
   */
  public static void main(String[] args) {

    try {
      System.out.println(Evaluation.evaluateModel(new Id3(), args));
    } catch (Exception e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Figure 8.11

weka.core.Instances to delete all instances with missing class values, because these instances are useless in the training process. Finally it calls makeTree(), which actually builds the decision tree by recursively generating all subtrees attached to the root node.

MakeTree()

In makeTree(), the first step is to check whether the dataset is empty. If not, a leaf is created by setting m_Attribute to null. The class value m_ClassValue assigned to this leaf is set to be missing, and the estimated probability for each of the dataset's classes in m_Distribution is initialized to zero. If training instances are present, makeTree() finds the attribute that yields the greatest information gain for them. It first creates a Java Enumeration of the dataset's attributes. If the index of the class attribute is set—as it will be for this dataset—the class is automatically excluded from the enumeration. Inside the enumeration, the information gain for each attribute is computed by computeInfoGain() and stored in an array. We will return to this method later. The index() method from weka.core.Attribute returns the attribute's index in the dataset, which is used to index the array. Once the enumeration is complete, the attribute with greatest information gain is stored in the class variable m_Attribute. The maxIndex() method from weka.core.Utils returns the index of the greatest value in an array of integers or doubles. (If there is more than one element with maximum value, the first is returned.) The index of this attribute is passed to the attribute() method from weka.core.Instances, which returns the corresponding attribute.

You might wonder what happens to the array field corresponding to the class attribute. We need not worry about this because Java automatically initializes all elements in an array of numbers to zero, and the information gain is always greater than or equal to zero. If the maximum information gain is zero, makeTree() creates a leaf. In that case m_Attribute is set to null, and makeTree() computes both the distribution of class probabilities and the class with greatest probability. (The normalize() method from weka.core.Utils normalizes an array of doubles so that its sum is 1.)

When it makes a leaf with a class value assigned to it, makeTree() stores the class attribute in m_ClassAttribute. This is because the method that outputs the decision tree needs to access this in order to print the class label.

If an attribute with nonzero information gain is found, makeTree() splits the dataset according to the attribute's values and recursively builds subtrees for each of the new datasets. To make the split it calls the method splitData(). This first creates as many empty datasets as there are attribute values and stores them in an array (setting the initial capacity of each dataset to the number of instances in the original dataset), then iterates through all instances in the original dataset and allocates them to the new

dataset that corresponds to the attribute's value. Returning to `makeTree()`, the resulting array of datasets is used for building subtrees. The method creates an array of `Id3` objects, one for each attribute value, and calls `buildClassifier()` on each by passing it the corresponding dataset.

ComputeInfoGain()

Returning to `computeInfoGain()`, this calculates the information gain associated with an attribute and a dataset using a straightforward implementation of the method in Section 4.3 (pp. 92–94). First it computes the entropy of the dataset. Then it uses `splitData()` to divide it into subsets, and calls `computeEntropy()` on each one. Finally it returns the difference between the former entropy and the weighted sum of the latter ones—the information gain. The method `computeEntropy()` uses the `log2()` method from `weka.core.Utils` to compute the logarithm (to base 2) of a number.

ClassifyInstance()

Having seen how ID3 constructs a decision tree, we now examine how it uses the tree structure to predict class values and probabilities. Let's first look at `classifyInstance()`, which predicts a class value for a given instance. In Weka, nominal class values—like the values of all nominal attributes—are coded and stored in `double` variables, representing the index of the value's name in the attribute declaration. We chose this representation in favor of a more elegant object-oriented approach to increase speed of execution and reduce storage requirements. In our implementation of ID3, `classifyInstance()` recursively descends the tree, guided by the instance's attribute values, until it reaches a leaf. Then it returns the class value `m_ClassValue` stored at this leaf. The method `distributionForInstance()` works in exactly the same way, returning the probability distribution stored in `m_Distribution`.

Most machine learning models, and in particular decision trees, serve as a more or less comprehensible explanation of the structure found in the data. Accordingly each of Weka's classifiers, like many other Java objects, implements a `toString()` method that produces a textual representation of itself in the form of a `String` variable. ID3's `toString()` method outputs a decision tree in roughly the same format as `J4.8` (Figure 8.2). It recursively prints the tree structure into a `String` variable by accessing the attribute information stored at the nodes. To obtain each attribute's name and values, it uses the `name()` and `value()` methods from `weka.core.Attribute`.

Main()

The only method in `Id3` that hasn't been described is `main()`, which is

called whenever the class is executed from the command line. As you can see, it's simple: it basically just tells Weka's `Evaluation` class to evaluate `Id3` with the given command-line options, and prints the resulting string. The one-line expression that does this is enclosed in a `try-catch` statement, which catches the various exceptions that can be thrown by Weka's routines or other Java methods.

The `evaluation()` method in `weka.classifiers.Evaluation` interprets the generic scheme-independent command-line options discussed in Section 8.3, and acts appropriately. For example, it takes the `-t` option, which gives the name of the training file, and loads the corresponding dataset. If no test file is given, it performs a cross-validation by repeatedly creating classifier objects and calling `buildClassifier()`, `classifyInstance()`, and `distributionForInstance()` on different subsets of the training data. Unless the user suppresses output of the model by setting the corresponding command-line option, it also calls the `toString()` method to output the model built from the full training dataset.

What happens if the scheme needs to interpret a specific option such as a pruning parameter? This is accomplished using the `OptionHandler` interface in `weka.classifiers`. A classifier that implements this interface contains three methods, `listOptions()`, `setOptions()`, and `getOptions()`, which can be used to list all the classifier's scheme-specific options, to set some of them, and to get the options that are currently set. The `evaluation()` method in `Evaluation` automatically calls these methods if the classifier implements the `OptionHandler` interface. Once the scheme-independent options have been processed, it calls `setOptions()` to process the remaining options before using `buildClassifier()` to generate a new classifier. When it outputs the classifier, it uses `getOptions()` to output a list of the options that are currently set. For a simple example of how to implement these methods, look at the source code for `weka.classifiers.OneR`.

Some classifiers are incremental, that is, they can be incrementally updated as new training instances arrive and don't have to process all the data in one batch. In Weka, incremental classifiers implement the `UpdateableClassifier` interface in `weka.classifiers`. This interface declares only one method, namely `updateClassifier()`, which takes a single training instance as its argument. For an example of how to use this interface, look at the source code for `weka.classifiers.IBk`.

If a classifier is able to make use of instance weights, it should implement the `WeightedInstancesHandler()` interface from *weka.core*. Then other algorithms, such as the boosting algorithms, can make use of this property.

Conventions for implementing classifiers

There are some conventions that you must obey when implementing classifiers in Weka. If you do not, things will go awry—for example,

Weka's evaluation module might not compute the classifier's statistics properly when evaluating it.

The first convention has already been mentioned: each time a classifier's `buildClassifier()` method is called, it must reset the model. The `CheckClassifier` class described in Section 8.3 performs appropriate tests to ensure that this is the case. When `buildClassifier()` is called on a dataset, the same result must always be obtained, regardless of how often the classifier has been applied before to other datasets. However, `buildClassifier()` must not reset class variables that correspond to scheme-specific options, because these settings must persist through multiple calls of `buildClassifier()`. Also, a call of `buildClassifier()` must never change the input data.

The second convention is that when the learning scheme can't make a prediction, the classifier's `classifyInstance()` method must return `Instance.missingValue()` and its `distributionForInstance()` method must return zero probabilities for all classes. The ID3 implementation in Figure 8.11 does this.

The third convention concerns classifiers for numeric prediction. If a `Classifier` is used for numeric prediction, `classifyInstance()` just returns the numeric value that it predicts. In some cases, however, a classifier might be able to predict nominal classes and their class probabilities, as well as numeric class values—`weka.classifiers.IBk` is an example. In that case, the classifier is a `DistributionClassifier` and implements the `distributionForInstance()` method. What should `distributionForInstance()` return if the class is numeric? Weka's convention is that it returns an array of size one whose only element contains the predicted numeric value.

Another convention—not absolutely essential, but very useful nonetheless—is that every classifier implements a `toString()` method that outputs a textual description of itself.

Writing filters

There are two kinds of filter algorithms in Weka, depending on whether, like `DiscretizeFilter`, they must accumulate statistics from the whole input dataset before processing any instances, or, like `AttributeFilter`, they can process each instance immediately. We present an implementation of the first kind, and point out the main differences from the second kind, which is simpler.

The `Filter` superclass contains several generic methods for filter construction, listed in Table 8.9, that are automatically inherited by its subclasses. Writing a new filter essentially involves overriding some of these. `Filter` also documents the purpose of these methods, and how they need to be changed for particular types of filter algorithm.

Table 8.9 Public methods in the Filter class.

method	description
`boolean inputFormat(Instances)`	Set input format of data, returning `true` if output format can be collected immediately
`Instances outputFormat()`	Return output format of data
`boolean input(Instance)`	Input instance into filter, returning `true` if instance can be output immediately
`boolean batchFinished()`	Inform filter that all training data has been input, returning `true` if instances are pending output
`Instance output()`	Output instance from the filter
`Instance outputPeek()`	Output instance without removing it from output queue
`int numPendingOutput()`	Return number of instances waiting for output
`boolean isOutputFormatDefined()`	Return `true` if output format can be collected

The first step in using a filter is to inform it of the input data format, accomplished by the method `inputFormat()`. This takes an object of class `Instances` and uses its attribute information to interpret future input instances. The filter's output data format can be determined by calling `outputFormat()`—also stored as an object of class `Instances`. For filters that process instances at once, the output format is determined as soon as the input format has been specified. However, for those that must see the whole dataset before processing any individual instance, the situation depends on the particular filter algorithm. For example, `DiscretizeFilter` needs to see all training instances before determining the output format, because the number of discretization intervals is determined by the data. Consequently the method `inputFormat()` returns `true` if the output format can be determined as soon as the input format has been specified, and `false` otherwise. Another way of checking whether the output format exists is to call `isOutputFormatDefined()`.

Two methods are used for piping instances through the filter: `input()` and `output()`. As its name implies, the former gets an instance into the filter; it returns `true` if the processed instance is available immediately and `false` otherwise. The latter outputs an instance from the filter and removes it from its output queue. The `outputPeek()` method outputs a filtered instance without removing it from the output queue, and the number of instances in the queue can be obtained with `numPendingOutput()`.

Filters that must see the whole dataset before processing instances need to be notified when all training instances have been input. This is done by calling `batchFinished()`, which tells the filter that the statistics obtained from the input data gathered so far—the training data—should not be

updated when further data is received. For all filter algorithms, once `batchFinished()` has been called, the output format can be read and the filtered training instances are ready for output. The first time `input()` is called after `batchFinished()`, the output queue is reset—that is, all training instances are removed from it. If there are training instances awaiting output, `batchFinished()` returns `true`, otherwise `false`.

An example filter

It's time for an example. The `ReplaceMissingValuesFilter` takes a dataset and replaces missing values with a constant. For numeric attributes, the constant is the attribute's mean value; for nominal ones, its mode. This filter must see all the training data before any output can be determined, and once these statistics have been computed. they must remain fixed when future test data is filtered. Figure 8.12 shows the source code.

InputFormat()

`ReplaceMissingValuesFilter` overwrites three of the methods defined in `Filter`: `inputFormat()`, `input()`, and `batchFinished()`. In `inputFormat()`, as you can see from Figure 8.12, a dataset `m_InputFormat` is created with the required input format and capacity zero; this will hold incoming instances. The method `setOutputFormat()`, which is a protected method in `Filter`, is called to set the output format. Then the variable `b_NewBatch`, which indicates whether the next incoming instance belongs to a new batch of data, is set to `true` because a new dataset is to be processed; and `m_ModesAndMeans`, which will hold the filter's statistics, is initialized. The variables `b_NewBatch` and `m_InputFormat` are the only fields declared in the superclass `Filter` that are visible in `ReplaceMissingValuesFilter`, and they must be dealt with appropriately. As you can see from Figure 8.12, the method `inputFormat()` returns `true` because the output format can be collected immediately—replacing missing values doesn't change the dataset's attribute information.

Input()

An exception is thrown in `input()` if the input format is not set. Otherwise, if `b_NewBatch` is `true`—that is, if a new batch of data is to be processed—the filter's output queue is initialized, causing all instances awaiting output to be deleted, and the flag `b_NewBatch` is set to `false`, because a new instance is about to be processed. Then, if statistics have not yet been accumulated from the training data (that is, if `m_ModesAndMeans` is null), the new instance is added to `m_InputFormat` and `input()` returns `false` because the instance is not yet available for output. Otherwise, the instance is converted using `convertInstance()`, and `true` is returned. The method `convertInstance()` transforms an instance to the output format by replacing missing values with the modes and means, and appends it to the output queue by calling

push(), a protected method defined in Filter. Instances in the filter's output queue are ready for collection by output().

BatchFinished()

In batchFinished(), the filter first checks whether the input format is defined. Then, if no statistics have been stored in m_ModesAndMeans by a previous call, the modes and means are computed and the training instances are converted using convertInstance(). Finally, regardless of the status of m_ModesAndMeans, b_NewBatch is set to true to indicate that the last batch has been processed, and true is returned if instances are available in the output queue.

Main()

The main() method evaluates the command-line options and applies the filter. It does so by calling two methods from Filter: batchFilterFile() and filterFile(). The former is called if a test file is provided as well as a training file (using the -b command-line option); otherwise the latter is called. Both methods interpret the command-line options. If the filter implements the OptionHandler interface, its setOptions(), getOptions(), and listOptions() methods are called to deal with any filter-specific options, just as in the case of classifiers.

In general, as in this particular example of ReplaceMissingValuesFilter, only the three routines inputFormat(), input(), and batchFinished() need be changed in order to implement a filter with new functionality. The method outputFormat() from Filter is actually declared final and can't be overwritten anyway. Moreover, if the filter can process each instance immediately, batchFinished() need not be altered—the default implementation will do the job. A simple (but not very useful) example of such a filter is weka.filters.AllFilter, which passes all instances through unchanged.

Conventions for writing filters

By now, most of the requirements for implementing filters should be clear. However, some deserve explicit mention. First, filters must never change the input data, nor add instances to the dataset used to provide the input format. ReplaceMissingValuesFilter() avoids this by storing an empty copy of the dataset in m_InputFormat. Second, calling inputFormat() should initialize the filter's internal state, but not alter any variables corresponding to user-provided command-line options. Third, instances input to the filter should never be pushed directly on to the output queue: they must be replaced by brand new objects of class Instance. Otherwise, anomalies will appear if the input instances are changed outside the filter later on. For example, AllFilter calls the copy() method in Instance to create a copy of each instance before pushing it on to the output queue.

```
import weka.filters.*;
import weka.core.*;
import java.io.*;

/**
 * Replaces all missing values for nominal and numeric attributes in a
 * dataset with the modes and means from the training data.
 */
public class ReplaceMissingValuesFilter extends Filter {

  /** The modes and means */
  private double[] m_ModesAndMeans = null;

  /**
   * Sets the format of the input instances.
   */
  public boolean inputFormat(Instances instanceInfo)
       throws Exception {

    m_InputFormat = new Instances(instanceInfo, 0);
    setOutputFormat(m_InputFormat);
    b_NewBatch = true;
    m_ModesAndMeans = null;
    return true;
  }

  /**
   * Input an instance for filtering. Filter requires all
   * training instances be read before producing output.
   */
  public boolean input(Instance instance) throws Exception {

    if (m_InputFormat == null) {
      throw new Exception("No input instance format defined");
    }
    if (b_NewBatch) {
      resetQueue();
      b_NewBatch = false;
    }
    if (m_ModesAndMeans == null) {
      m_InputFormat.add(instance);
      return false;
    } else {
      convertInstance(instance);
      return true;
    }
  }

  /**
   * Signify that this batch of input to the filter is finished.
   */

  public boolean batchFinished() throws Exception {

    if (m_InputFormat == null) {
      throw new Exception("No input instance format defined");
    }
```

Figure 8.11 Source code for the ID3 decision tree learner.

```
    if (m_ModesAndMeans == null) {

      // Compute modes and means
      m_ModesAndMeans = new double[m_InputFormat.numAttributes()];
      for (int i = 0; i < m_InputFormat.numAttributes(); i++) {
        if (m_InputFormat.attribute(i).isNominal() ||
            m_InputFormat.attribute(i).isNumeric()) {
          m_ModesAndMeans[i] = m_InputFormat.meanOrMode(i);
        }
      }

      // Convert pending input instances
      for(int i = 0; i < m_InputFormat.numInstances(); i++) {
        Instance current = m_InputFormat.instance(i);
        convertInstance(current);
      }
    }

    b_NewBatch = true;
    return (numPendingOutput() != 0);
  }

  /**
   * Convert a single instance over. The converted instance is
   * added to the end of the output queue.
   */
  private void convertInstance(Instance instance) throws Exception {

    Instance newInstance = new Instance(instance);

    for(int j = 0; j < m_InputFormat.numAttributes(); j++){
      if (instance.isMissing(j) &&
          (m_InputFormat.attribute(j).isNominal() ||
           m_InputFormat.attribute(j).isNumeric())) {
        newInstance.setValue(j, m_ModesAndMeans[j]);
      }
    }
    push(newInstance);
  }

  /**
   * Main method.
   */
  public static void main(String [] argv) {

    try {
      if (Utils.getFlag('b', argv)) {
         Filter.batchFilterFile(new ReplaceMissingValuesFilter(),argv);
      } else {
        Filter.filterFile(new ReplaceMissingValuesFilter(),argv);
      }
    } catch (Exception ex) {
      System.out.println(ex.getMessage());
    }
  }
}
```

Figure 8.11

This tutorial is Chapter 8 of the book *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Cross-references are to other sections of that book.