CHAPTER

The Command-Line Interface

14

Lurking behind Weka's interactive interfaces—the Explorer, the Knowledge Flow, and the Experimenter—lies its basic functionality. This can be accessed more directly through a command-line interface. Select *Simple CLI* from the interface choices at the right of Figure 11.3(a) to bring up a plain textual panel with a line at the bottom on which you enter commands. Alternatively, use the operating system's command-line interface to run the classes in *weka.jar*, in which case you must first set the *CLASSPATH* environment variable as explained in Weka's *README* file.

14.1 GETTING STARTED

At the beginning of Section 11.1 (page 410) we used the Explorer to invoke the J4.8 learner on the weather data. To do the same thing in the command-line interface, type the following into the line at the bottom of the text panel:

```
java weka.classifiers.trees.J48 -t data/weather.arff
```

This incantation calls the Java virtual machine (in the Simple CLI, Java is already loaded) and instructs it to execute J4.8. Weka is organized in *packages* that correspond to a directory hierarchy. The program to be executed is called *J48* and resides in the *trees* package, which is a subpackage of *classifiers*, which is part of the overall *weka* package. The next section gives more details on the package structure. The *–t* option signals that the next argument is the name of the training file: We are assuming that the weather data resides in a *data* subdirectory of the directory from which you fired up Weka. The result resembles the text shown in Figure 11.5. In the Simple CLI it appears in the panel above the line where you typed the command.

14.2 THE STRUCTURE OF WEKA

We have explained how to invoke filtering and learning schemes with the Explorer interface and connect them together with the Knowledge Flow interface. To go further, it is necessary to learn something about how Weka is put together. Detailed, up-to-date information can be found in the online documentation included in the distribution.

This is more technical than the descriptions of the learning and filtering schemes given by the *More* button in the Explorer and Knowledge Flow's object editors. It 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 or collection of classes. In object-oriented programming, a *class* is a collection of variables along with some *methods* that operate on them. 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 in this book, where the terms *class* and *instance* appear in the quite different context of machine learning. From now on, you will have to infer the intended meaning of these terms from their context. This is not difficult—and 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 encapsulated in a class, and it may depend on other classes for some of its functionality. For example, the *J48* class described previously 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.

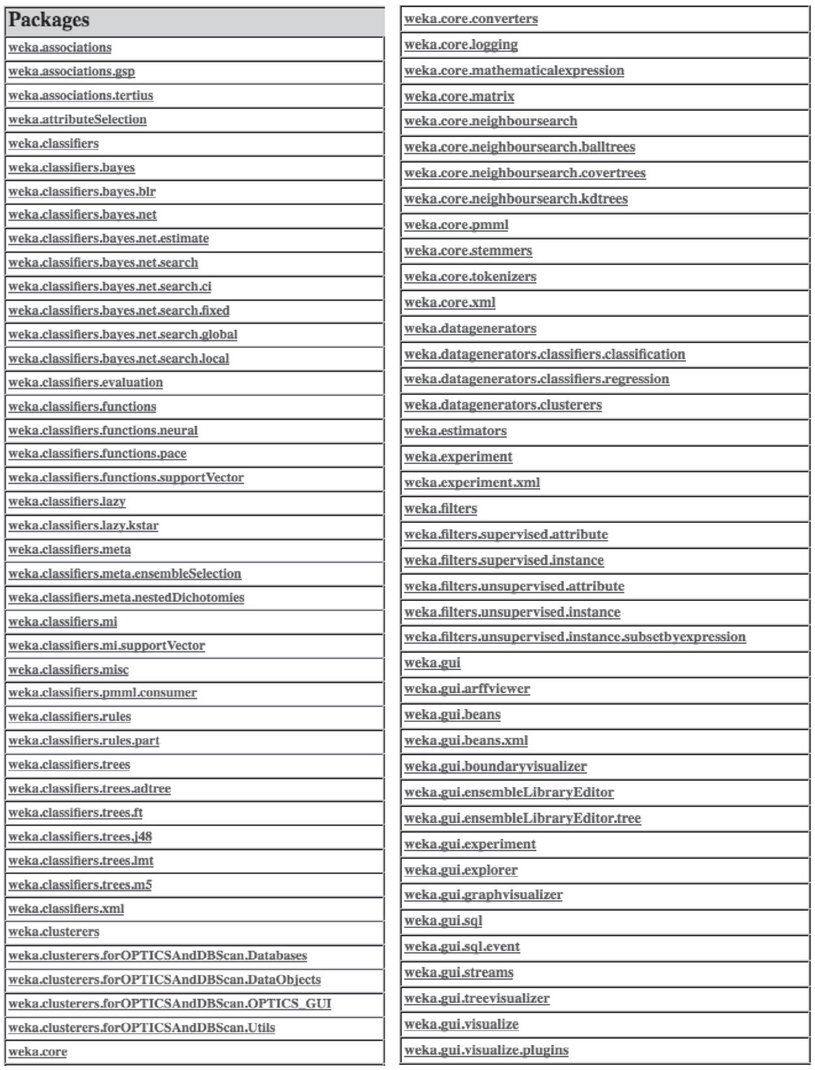
Larger programs are usually split into more than one class. The *J48* class, for example, does not actually contain any code for building a decision tree. It includes references to instances of other classes that do most of the work. When there are a lot of classes—as in Weka—they become difficult to comprehend and navigate. Java allows classes to be organized into packages. A *package* is just a directory containing a collection of related classes; for example, the *trees* package mentioned previously contains the classes that implement decision trees. Packages are organized in a hierarchy that corresponds to the directory hierarchy: *trees* 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 an alphabetical list of all the packages in Weka, as shown in Figure 14.1(a). (If you view the Javadoc with frames, you will see more than this. Click on *NO FRAMES* to remove the extra information.) Here we introduce a few of them in order of importance.

The *weka.core* Package

The *core* package is central to the Weka system, and its classes are accessed from almost every other class. You can determine what they are by clicking on the *weka.core* hyperlink, which brings up the web page shown in Figure 14.1(b).

This web page is divided into several parts, the main ones being the *interface summary* and the *class summary*. The latter is a list of classes contained within the package, and the former lists the interfaces it provides. An interface is 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

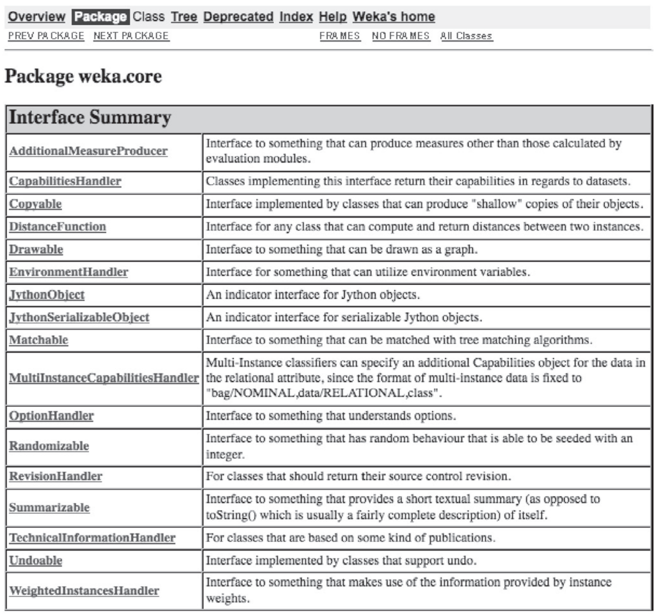


Packages
weka.associations
weka.associations.gsp
weka.associations.tertius
weka.attributeSelection
weka.classifiers
weka.classifiers.bayes
weka.classifiers.bayes.blr
weka.classifiers.bayes.net
weka.classifiers.bayes.net.estimate
weka.classifiers.bayes.net.search
weka.classifiers.bayes.net.search.ci
weka.classifiers.bayes.net.search.fixed
weka.classifiers.bayes.net.search.global
weka.classifiers.bayes.net.search.local
weka.classifiers.evaluation
weka.classifiers.functions
weka.classifiers.functions.neural
weka.classifiers.functions.pace
weka.classifiers.functions.supportVector
weka.classifiers.lazy
weka.classifiers.lazy.kstar
weka.classifiers.meta
weka.classifiers.meta.ensembleSelection
weka.classifiers.meta.nestedDichotomies
weka.classifiers.mi
weka.classifiers.mi.supportVector
weka.classifiers.misc
weka.classifiers.pmml.consumer
weka.classifiers.rules
weka.classifiers.rules.part
weka.classifiers.trees
weka.classifiers.trees.adtree
weka.classifiers.trees.ft
weka.classifiers.trees.j48
weka.classifiers.trees.lmt
weka.classifiers.trees.m5
weka.classifiers.xml
weka.clusterers
weka.clusterers.forOPTICSAndDBScan.Databases
weka.clusterers.forOPTICSAndDBScan.DataObjects
weka.clusterers.forOPTICSAndDBScan.OPTICS_GUI
weka.clusterers.forOPTICSAndDBScan.Utils
weka.core
weka.core.converters
weka.core.logging
weka.core.mathematicalexpression
weka.core.matrix
weka.core.neighboursearch
weka.core.neighboursearch.balltrees
weka.core.neighboursearch.covertrees
weka.core.neighboursearch.kdtrees
weka.core.pmml
weka.core.stemmers
weka.core.tokenizers
weka.core.xml
weka.datagenerators
weka.datagenerators.classifiers.classification
weka.datagenerators.classifiers.regression
weka.datagenerators.clusterers
weka.estimators
weka.experiment
weka.experiment.xml
weka.filters
weka.filters.supervised.attribute
weka.filters.supervised.instance
weka.filters.unsupervised.attribute
weka.filters.unsupervised.instance
weka.filters.unsupervised.instance.subsetbyexpression
weka.gui
weka.gui.arffviewer
weka.gui.beans
weka.gui.beans.xml
weka.gui.boundaryvisualizer
weka.gui.ensembleLibraryEditor
weka.gui.ensembleLibraryEditor.tree
weka.gui.experiment
weka.gui.explorer
weka.gui.graphvisualizer
weka.gui.sql
weka.gui.sql.event
weka.gui.streams
weka.gui.treevisualizer
weka.gui.visualize
weka.gui.visualize.plugins

(a)

FIGURE 14.1

Using Javadoc: (a) the front page and (b) the *weka.core* package (see next page).

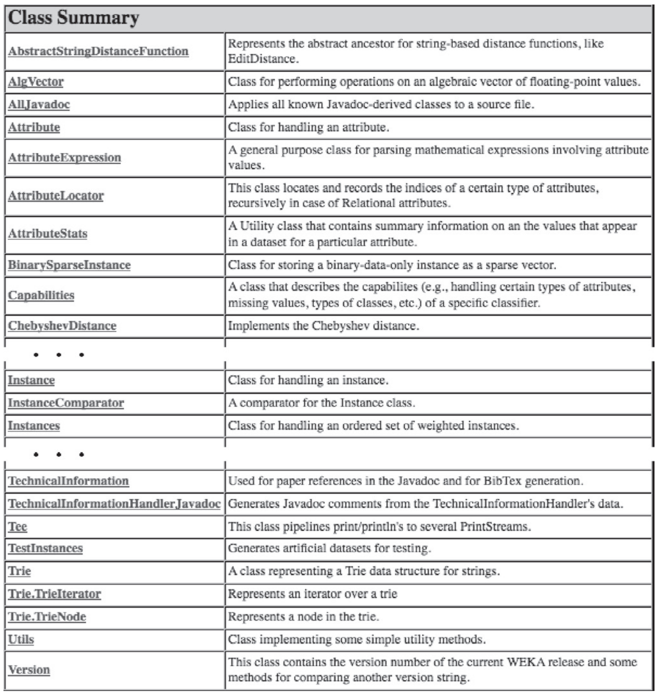


Overview **Package** Class Tree Deprecated Index Help Weka's home
PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES All Classes

Package weka.core

Interface Summary	
AdditionalMeasureProducer	Interface to something that can produce measures other than those calculated by evaluation modules.
CapabilitiesHandler	Classes implementing this interface return their capabilities in regards to datasets.
Copyable	Interface implemented by classes that can produce "shallow" copies of their objects.
DistanceFunction	Interface for any class that can compute and return distances between two instances.
Drawable	Interface to something that can be drawn as a graph.
EnvironmentHandler	Interface for something that can utilize environment variables.
JythonObject	An indicator interface for Jython objects.
JythonSerializableObject	An indicator interface for serializable Jython objects.
Matchable	Interface to something that can be matched with tree matching algorithms.
MultiInstanceCapabilitiesHandler	Multi-Instance classifiers can specify an additional Capabilities object for the data in the relational attribute, since the format of multi-instance data is fixed to "bag/NOMINAL,data/RELATIONAL,class".
OptionHandler	Interface to something that understands options.
Randomizable	Interface to something that has random behaviour that is able to be seeded with an integer.
RevisionHandler	For classes that should return their source control revision.
Summarizable	Interface to something that provides a short textual summary (as opposed to toString() which is usually a fairly complete description) of itself.
TechnicalInformationHandler	For classes that are based on some kind of publications.
Undoable	Interface implemented by classes that support undo.
WeightedInstancesHandler	Interface to something that makes use of the information provided by instance weights.

(b1)



Class Summary	
AbstractStringDistanceFunction	Represents the abstract ancestor for string-based distance functions, like EditDistance.
AlgVector	Class for performing operations on an algebraic vector of floating-point values.
AllJavadoc	Applies all known Javadoc-derived classes to a source file.
Attribute	Class for handling an attribute.
AttributeExpression	A general purpose class for parsing mathematical expressions involving attribute values.
AttributeLocator	This class locates and records the indices of a certain type of attributes, recursively in case of Relational attributes.
AttributeStats	A Utility class that contains summary information on an the values that appear in a dataset for a particular attribute.
BinarySparseInstance	Class for storing a binary-data-only instance as a sparse vector.
Capabilities	A class that describes the capabilites (e.g., handling certain types of attributes, missing values, types of classes, etc.) of a specific classifier.
ChebyshevDistance	Implements the Chebyshev distance.
• • •	
Instance	Class for handling an instance.
InstanceComparator	A comparator for the Instance class.
Instances	Class for handling an ordered set of weighted instances.
• • •	
TechnicalInformation	Used for paper references in the Javadoc and for BibTex generation.
TechnicalInformationHandlerJavadoc	Generates Javadoc comments from the TechnicalInformationHandler's data.
Tee	This class pipelines print/println's to several PrintStreams.
TestInstances	Generates artificial datasets for testing.
Trie	A class representing a Trie data structure for strings.
Trie.TrieIterator	Represents an iterator over a trie
Trie.TrieNode	Represents a node in the trie.
Utils	Class implementing some simple utility methods.
Version	This class contains the version number of the current WEKA release and some methods for comparing another version string.

(b2)

FIGURE 14.1, cont'd

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 *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 or string attribute, its possible values. An object of class *Instance* contains the attribute values of a particular instance; an object of class *Instances* holds an ordered set of instances—in other words, a dataset. You can learn more about these classes by clicking their hyperlinks; we return to them in Chapter 15 page 536 and 538) when we show how to invoke machine learning schemes from other Java code. However, you can use Weka from the command-line interface without knowing the details.

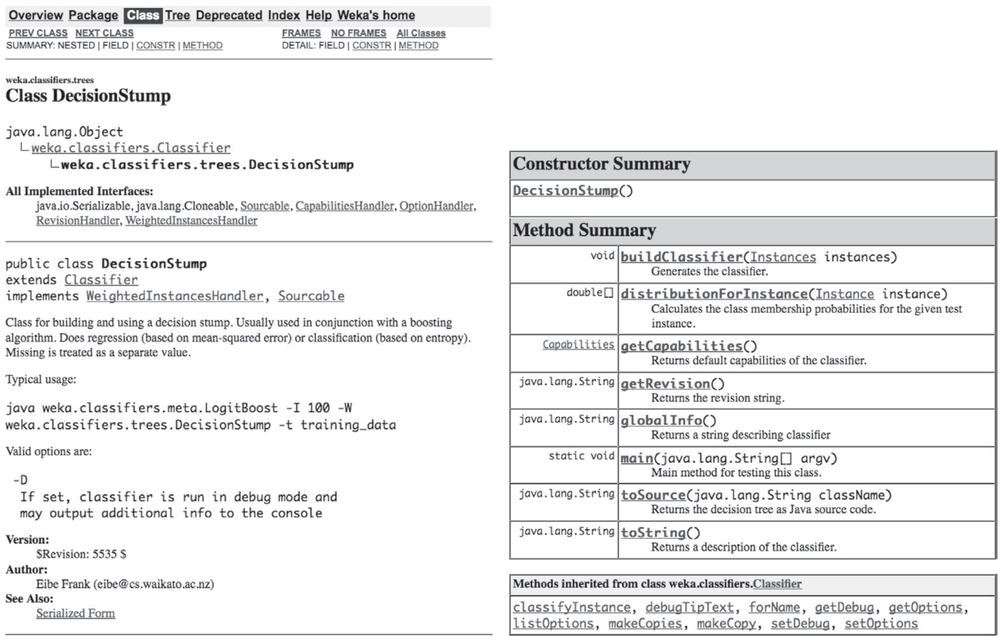
Clicking the *Overview* hyperlink in the upper left corner of any documentation page returns you to the listing of all the packages in Weka, shown in Figure 14.1(a).

The *weka.classifiers* Package

The *classifiers* package contains implementations of most of the algorithms for classification and numeric prediction described 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. *Classifier* contains three methods: *buildClassifier*(), *classifyInstance*(), and *distributionForInstance*(). In the terminology of object-oriented programming, the learning algorithms are represented by subclasses of *Classifier* and therefore automatically inherit these three 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. Thus, for example, the same evaluation module can be used to evaluate the performance of any classifier in Weka.

To see an example, click on *weka.classifiers.trees* and then 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 14.2, shows the fully qualified name of this class, *weka.classifiers.trees.DecisionStump*, near the top. You have to use this rather lengthy name whenever you build a decision stump from the command line. The class name is sited in a small tree structure showing the relevant part of the class hierarchy. As you can see, *DecisionStump* is a subclass of *weka.classifiers.Classifier*, which is itself a subclass of *java.lang.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—brief documentation, its version, and the author—Figure 14.2 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,



Overview Package Class Tree Deprecated Index Help Weka's home
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

weka.classifiers.trees
Class DecisionStump

```
java.lang.Object
  └weka.classifiers.Classifier
      └weka.classifiers.trees.DecisionStump
```

All Implemented Interfaces:
java.io.Serializable, java.lang.Cloneable, Sourcable, CapabilitiesHandler, OptionHandler, RevisionHandler, WeightedInstancesHandler

```
public class DecisionStump
extends Classifier
implements WeightedInstancesHandler, Sourcable
```

Class for building and using a decision stump. Usually used in conjunction with a boosting algorithm. Does regression (based on mean-squared error) or classification (based on entropy). Missing is treated as a separate value.

Typical usage:

```
java weka.classifiers.meta.LogitBoost -I 100 -W
weka.classifiers.trees.DecisionStump -t training_data
```

Valid options are:

```
 -D
  If set, classifier is run in debug mode and
  may output additional info to the console
```

Version:
$Revision: 5535 $
Author:
Eibe Frank (eibe@cs.waikato.ac.nz)
See Also:
Serialized Form

Constructor Summary
`DecisionStump()`

Method Summary	
`void`	`buildClassifier(Instances instances)` Generates the classifier.
`double[]`	`distributionForInstance(Instance instance)` Calculates the class membership probabilities for the given test instance.
`Capabilities`	`getCapabilities()` Returns default capabilities of the classifier.
`java.lang.String`	`getRevision()` Returns the revision string.
`java.lang.String`	`globalInfo()` Returns a string describing classifier
`static void`	`main(java.lang.String[] argv)` Main method for testing this class.
`java.lang.String`	`toSource(java.lang.String className)` Returns the decision tree as Java source code.
`java.lang.String`	`toString()` Returns a description of the classifier.

Methods inherited from class weka.classifiers.Classifier
`classifyInstance, debugTipText, forName, getDebug, getOptions, listOptions, makeCopies, makeCopy, setDebug, setOptions`

(a) (b)

FIGURE 14.2

DecisionStump, a class of the *weka.classifiers.trees* package.

and a short description of its functionality. Beneath those indexes, the web page gives more details about the constructors and methods. We return to these details later.

As you can see, *DecisionStump* overwrites the *distributionForInstance*() method from *Classifier*; the default implementation of *classifyInstance*() in *Classifier* then uses this method to produce its classifications. In addition, it contains the methods *getCapabilities*(), *getRevision*(), *globalInfo*(), *toSource*(), *toString*(), and *main*(). We discuss *getCapabilities*() shortly. The *getRevision*() method simply returns the revision number of the classifier. There is a utility class in the *weka.core* package that prints it to the screen; it is used by Weka maintainers when diagnosing and debugging problems reported by users. The *globalInfo*() method returns a string describing the classifier, which, along with the scheme's options, is displayed by the *More* button in the generic object editor (see Figure 11.7(b)). The *toString*() method returns a textual representation of the classifier, used whenever it is printed on the screen, while the *toSource*() method is used to obtain a source code representation of the learned classifier. The *main*() method is called when you ask for a decision stump from the command line—in other words, every time you enter a command beginning with

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

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

The *getCapabilities*() method is called by the generic object editor to provide information about the capabilities of a learning scheme (Figure 11.9(d)). The training data is checked against the learning scheme's capabilities when the *buildClassifier*() method is called, and an error is raised when the classifier's stated capabilities do not match the data's characteristics. The *getCapabilities*() method is present in the *Classifier* class and, by default, enables all capabilities (i.e., imposes no constraints). This makes it easier for new Weka programmers to get started because they need not learn about and specify capabilities initially. Capabilities are covered in more detail in Chapter 16 (page 555).

Other Packages

Several other packages listed earlier in Figure 14.1(a) are worth mentioning: *weka.associations*, *weka.clusterers*, *weka.datagenerators*, *weka.estimators*, *weka.filters*, and *weka.attributeSelection*. The *weka.associations* package contains association-rule learners. These learners have been placed in a separate package because association rules are fundamentally different from classifiers. The *weka.clusterers* package contains methods for unsupervised learning. Artificial data can be generated using the classes in *weka.datagenerators*. The *weka.estimators* package contains subclasses of a generic *Estimator* class, which computes different types of probability distribution. These subclasses are used by the Naïve Bayes algorithm (among others).

In the *weka.filters* package, the *Filter* class defines the general structure of classes containing filter algorithms, which are all implemented as subclasses of *Filter*. Like classifiers, filters can be used from the command line—we will see how shortly. The *weka.attributeSelection* package contains several classes for attribute selection. The classes are used by the *AttributeSelectionFilter* in *weka.filters.supervised.attribute*, but can also be invoked separately.

Javadoc Indexes

As mentioned previously, all classes are automatically subclasses of *Object*. To examine the tree that corresponds to Weka's hierarchy of classes, select the *Overview* link from the top of any page of the online documentation. Click *Tree* to display the overview as a tree that shows 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 of the classes, packages, 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 *Overview* and then *Index*.

Suppose you want to check which Weka classifiers and filters are capable of operating incrementally. Searching for the word *incremental* in the index would soon lead you to the keyword *UpdateableClassifier*. In fact, this is a Java interface; interfaces are listed after the classes in the overview tree. You are looking for all classes that implement this interface. Clicking any occurrence of it in the documentation brings up a page that describes the interface and lists the classifiers that implement it. To find the filters is a little trickier unless you know the keyword *StreamableFilter*, which is the name of the interface that streams data through a filter; again, its page lists the filters that implement it. You would stumble across that keyword if you knew any example of a filter that could operate incrementally.

14.3 COMMAND-LINE OPTIONS

In the preceding example, the *–t* option was used on the command line to communicate the name of the training file to the learning algorithm. There are many 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 with the *–h* or *–help* option, or without any command-line options at all, it displays the applicable options: first the general options, then the scheme-specific ones. In the command-line interface, type

```
java weka.classifiers.trees.J48 –h
```

You'll see a list of the options common to all learning schemes, shown in Table 14.1, followed by those that apply only to *J48*, shown in Table 14.2. A notable one is *–info*, which outputs a very brief description of the scheme. We will explain the generic options and then briefly review the scheme-specific ones.

Generic Options

The options in Table 14.1 determine which data is used for training and testing, how the classifier is evaluated, and what kind of statistics are displayed. For example, the *–T* option is used to provide the name of the test file when evaluating a learning scheme on an independent test set. By default, the class is the last attribute in an ARFF file, but you can declare another one to be the class using *–c* followed by the position of the desired attribute—1 for the first, 2 for the second, and so on.

When cross-validation is performed (the default if a test file is not provided), the data is randomly shuffled first. To repeat the cross-validation several times, each time reshuffling the data in a different way, 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*. If performance on the training data alone is required, *–no-cv* can be used to suppress cross-validation; *–v* suppresses output of performance on the training data. As an alternative to cross-validation, a train-test split of the data specified with the *–t* option can be performed

Table 14.1 Generic Options for Learning Schemes in Weka

Option	Function
h or –help	Print help information
–synopsis or –info	In combination with –h or –help, prints the information from the "More" button in a classifier's generic object editor
–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
–no-cv	Don't perform cross-validation
–split-percentage <training percentage>	Specify percentage of the data to use for the training set in a train-test split
–preserve-order	Preserve original order of the data when performing a train-test split
–m <cost matrix file>	Specify file containing cost matrix
–l <input file>	Specify input file for model
–d <output file>	Specify output file for model
–v	Output no statistics for training data
–o	Output statistics only, not the classifier
–i	Output information retrieval statistics for two-class problems
–k	Output information-theoretic statistics
–p <attribute range>	Output predictions for test instances
–distribution	In combination with –*p*, output the full probability distribution for discrete class data instead of just the predicted label
–r	Output cumulative margin distribution
–z <class name>	Output the source representation of the classifier
–g	Output the graph representation of the classifier
–xml <filename> | <xml string>	Set scheme-specific options from XML-encoded options stored in a file or in a supplied string
–threshold-file <file>	Save threshold data (for ROC curves, etc.) to a file
–threshold-label <label>	Class label for the threshold data

Table 14.2 Scheme-Specific Options for the *J48* 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 any 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
-B	Use binary splits only
-S	Don't perform subtree raising
-L	Retain instance information
-A	Smooth probability estimates using Laplace smoothing
-Q	Seed for shuffling data

by supplying a percentage to use as the new training set with *–split-percentage* (the remaining data is used as the test set). Randomization of the data can be suppressed when performing a train-test split by specifying *–preserve-order*.

In the Explorer interface, cost-sensitive evaluation is invoked as described in Section 11.1 (page 403). To achieve the same effect from the command line, use the *–m* option to provide the name of a file containing the cost matrix. Here is a cost matrix for the weather data:

```
2  2    % Number of rows and columns in the matrix
0  10   % If true class yes and prediction no, penalty is 10
1  0    % If true class no and prediction yes, penalty is 1
```

The first line gives the number of rows and columns—that is, the number of class values. Then comes the matrix of penalties. Comments introduced by % can be appended to the end of any line.

It is also possible to save and load models. If you provide the name of an output file using *–d*, Weka saves the classifier generated from the training data. To evaluate the same classifier on a new batch of test data, you load it back using *–l* instead of rebuilding it. If the classifier can be updated incrementally, 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 wish only to assess the performance of a learning scheme, use *–o* to suppress output of the model. Use *–i* to see the performance measures of precision, recall, *F*-measure, and area under the ROC curve (see Section 5.7, page 172). Use *–k* to compute information-theoretic measures from the probabilities derived by a learning scheme (see Section 5.6, page 161).

People often want to know which class values the learning scheme actually predicts for each test instance. The *–p* option prints each test instance's number, the index of its class value and the actual value, the index of the predicted class value and the predicted value, a "+" if the class was misclassified, and the probability of the predicted class value. The probability predicted for each of the possible class labels of an instance can be output by using the *–distribution* flag in conjunction with *–p*. In this case, "*" is placed beside the probability in the distribution that corresponds to the predicted class value. The *–p* option also outputs attribute values for each instance and must be followed by a specification of the range (e.g., 1–2)—use 0 if you don't want any attribute values. You can also output the *cumulative margin distribution* for the training data, which shows the distribution of the margin measure (see Section 8.4, page 361). Finally, you can output the classifier's source representation, and a graphical representation if the classifier can produce one.

Data relating to performance graphs such as ROC and recall–precision curves can be sent to a file using the *–threshold-file* option. The class label to treat as the positive class for generating the data can be specified with *–threshold-label*. The next section discusses how scheme-specific options are supplied on the command line; they can also be set from an XML file or string using the *–xml* option.

Scheme-Specific Options

Table 14.2 shows the options specific to *J48*. You can force the algorithm to use the unpruned tree instead of the pruned one. You can suppress subtree raising, which increases efficiency. You can set the confidence threshold for pruning and the minimum number of instances permissible at any leaf—both parameters were described in Section 6.1 (page 201). As well as C4.5's standard pruning procedure, reduced-error pruning (see Section 6.2, page 206) can be performed. The *–N* option governs the size of the holdout set: The dataset is divided equally into that number of parts and the last is held out (default value 3). You can smooth the probability estimates using the Laplace technique, set the random number seed for shuffling the data when selecting a pruning set, and store the instance information for future visualization. Finally, to build a binary tree instead of one with multiway branches for nominal attributes, use *–B*.