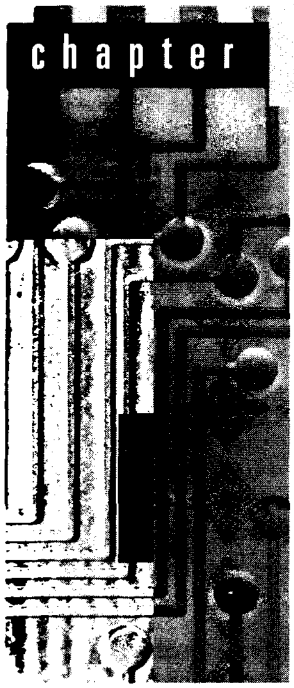


chapter 11 The Knowledge Flow Interface

With the Knowledge Flow interface, users select Weka components from a tool bar, place them on a layout canvas, and connect them into a directed graph that processes and analyzes data. It provides an alternative to the Explorer for those who like thinking in terms of how data flows through the system. It also allows the design and execution of configurations for streamed data processing, which the Explorer cannot do. You invoke the Knowledge Flow interface by selecting *KnowledgeFlow* from the choices at the bottom of the panel shown in Figure 10.3(a).

11.1 Getting started

Here is a step-by-step example that loads an ARFF file and performs a cross-validation using J4.8. We describe how to build up the final configuration shown in Figure 11.1. First create a source of data by clicking on the *DataSources* tab (rightmost entry in the bar at the top) and selecting *ARFFLoader* from the

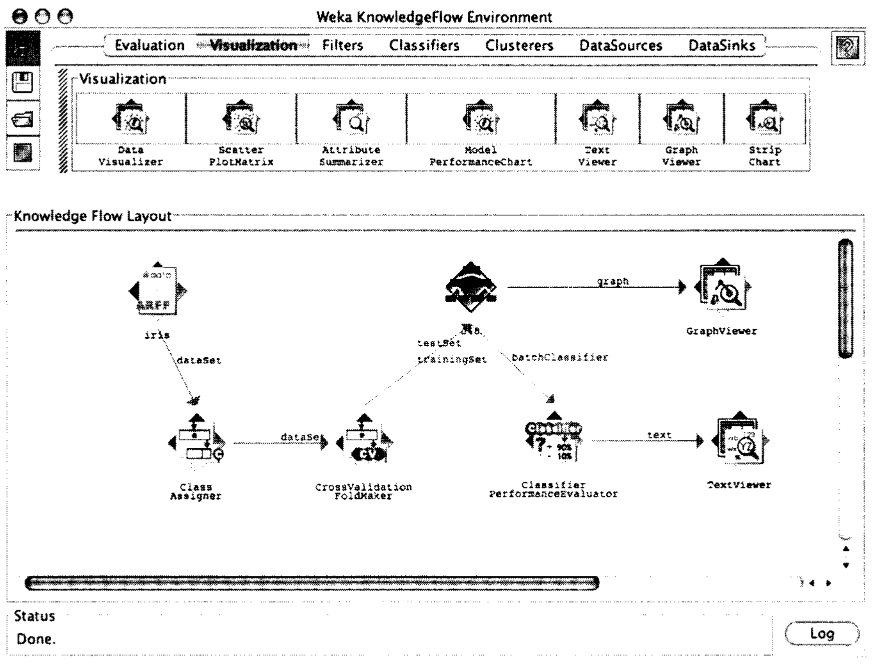
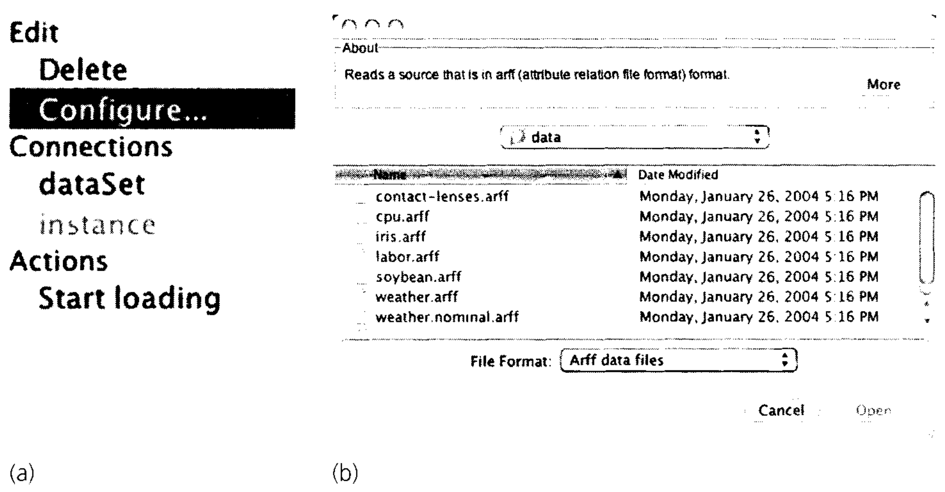


Figure 11.1 The Knowledge Flow interface.

toolbar. The mouse cursor changes to crosshairs to signal that you should next place the component. Do this by clicking anywhere on the canvas, whereupon a copy of the ARFF loader icon appears there. To connect it to an ARFF file, right-click it to bring up the pop-up menu shown in Figure 11.2(a). Click *Configure* to get the file browser in Figure 11.2(b), from which you select the desired ARFF file. The *File Format* pull-down menu allows you to choose a different type of data source—for example, spreadsheet files.

Now we specify which attribute is the class using a *ClassAssigner* object. This is on the *Evaluation* panel, so click the *Evaluation* tab, select the *ClassAssigner,* and place it on the canvas. To connect the data source to the class assigner, right-click the data source icon and select *dataset* from the menu, as shown in Figure 11.2(a). A rubber-band line appears. Move the mouse over the class assigner component and left-click. A red line labeled *dataset* appears, joining the two components. Having connected the class assigner, choose the class by right-clicking it, selecting *Configure,* and entering the location of the class attribute.

We will perform cross-validation on the *J48* classifier. In the data flow model, we first connect the *CrossValidationFoldMaker* to create the folds on which the classifier will run, and then pass its output to an object representing *J48. CrossValidationFoldMaker* is on the *Evaluation* panel. Select it, place it on the canvas, and connect it to the class assigner by right-clicking the latter and selecting



Reads a source that is in arff (attribute relation file format) format.

```
Name                      Date Modified
contact-lenses.arff       Monday, January 26, 2004 5:16 PM
cpu.arff                  Monday, January 26, 2004 5:16 PM
iris.arff                 Monday, January 26, 2004 5:16 PM
labor.arff                Monday, January 26, 2004 5:16 PM
soybean.arff              Monday, January 26, 2004 5:16 PM
weather.arff              Monday, January 26, 2004 5:16 PM
weather.nominal.arff      Monday, January 26, 2004 5:16 PM
```

Figure 11.2 Configuring a data source: (a) the right-click menu and (b) the file browser obtained from the *Configure* menu item.

dataset from the menu (which is similar to that in Figure 11.2(a)). Next select *J48* from the *Classifiers* panel and place a *J48* component on the canvas. There are so many different classifiers that you have to scroll along the toolbar to find it. Connect *J48* to the cross-validation fold maker in the usual way, but make the connection *twice* by first choosing *trainingSet* and then choosing *testSet* from the pop-up menu for the cross-validation fold maker. The next step is to select a *ClassifierPerformanceEvaluator* from the *Evaluation* panel and connect *J48* to it by selecting the *batchClassifier* entry from the pop-up menu for *J48*. Finally, from the *Visualization* toolbar we place a *TextViewer* component on the canvas. Connect the classifier performance evaluator to it by selecting the *text* entry from the pop-up menu for the performance evaluator.

At this stage the configuration is as shown in Figure 11.1 except that there is as yet no graph viewer. Start the flow of execution by selecting *Start loading* from the pop-up menu for the ARFF loader, shown in Figure 11.2(a). For a small dataset things happen quickly, but if the input were large you would see that some of the icons are animated—for example, *J48*'s tree would appear to grow and the performance evaluator's checkmarks would blink. Progress information appears in the status bar at the bottom of the interface. Choosing *Show results* from the text viewer's pop-up menu brings the results of cross-validation up in a separate window, in the same form as for the Explorer.

To complete the example, add a *GraphViewer* and connect it to *J48*'s *graph* output to see a graphical representation of the trees produced for each fold of the cross-validation. Once you have redone the cross-validation with this extra component in place, selecting *Show results* from its pop-up menu produces a list of trees, one for each cross-validation fold. By creating cross-validation folds and passing them to the classifier, the Knowledge Flow model provides a way to hook into the results for each fold. The Explorer cannot do this: it treats cross-validation as an evaluation method that is applied to the output of a classifier.

11.2 The Knowledge Flow components

Most of the Knowledge Flow components will be familiar from the Explorer. The *Classifiers* panel contains all of Weka's classifiers, the *Filters* panel contains the filters, and the *Clusterers* panel holds the clusterers. Possible data sources are ARFF files, CSV files exported from spreadsheets, the C4.5 file format, and a serialized instance loader for data files that have been saved as an instance of a Java object. There are data sinks and sources for the file formats supported by the Explorer. There is also a data sink and a data source that can connect to a database.

The components for visualization and evaluation, listed in Table 11.1, have not yet been encountered. Under *Visualization*, the *DataVisualizer* pops up a panel for visualizing data in a two-dimensional scatter plot as in Figure 10.6(b), in which you can select the attributes you would like to see. *ScatterPlotMatrix* pops up a matrix of two-dimensional scatter plots for every pair of attributes,

Table 11.1 Visualization and evaluation components.

	Name	Function
Visualization	*DataVisualizer*	Visualize data in a 2D scatter plot
	ScatterPlotMatrix	Matrix of scatter plots
	AttributeSummarizer	Set of histograms, one for each attribute
	ModelPerformanceChart	Draw ROC and other threshold curves
	TextViewer	Visualize data or models as text
	GraphViewer	Visualize tree-based models
	StripChart	Display a scrolling plot of data
Evaluation	*TrainingSetMaker*	Make a dataset into a training set
	TestSetMaker	Make a dataset into a test set
	CrossValidationFoldMaker	Split a dataset into folds
	TrainTestSplitMaker	Split a dataset into training and test sets
	ClassAssigner	Assign one of the attributes to be the class
	ClassValuePicker	Choose a value for the *positive* class
	ClassifierPerformanceEvaluator	Collect evaluation statistics for batch evaluation
	IncrementalClassifierEvaluator	Collect evaluation statistics for incremental evaluation
	ClustererPerformanceEvaluator	Collect evaluation statistics for clusterers
	PredictionAppender	Append a classifier's predictions to a dataset

shown in Figure 10.16(a). *AttributeSummarizer* gives a matrix of histograms, one for each attribute, like that in the lower right-hand corner of Figure 10.3(b). *ModelPerformanceChart* draws ROC curves and other threshold curves. *GraphViewer* pops up a panel for visualizing tree-based models, as in Figure 10.6(a). As before, you can zoom, pan, and visualize the instance data at a node (if it has been saved by the learning algorithm).

StripChart is a new visualization component designed for use with incremental learning. In conjunction with the *IncrementalClassifierEvaluator* described in the next paragraph it displays a learning curve that plots accuracy—both the percentage accuracy and the root mean-squared probability error—against time. It shows a fixed-size time window that scrolls horizontally to reveal the latest results.

The *Evaluation* panel has the components listed in the lower part of Table 11.1. The *TrainingSetMaker* and *TestSetMaker* make a dataset into the corresponding kind of set. The *CrossValidationFoldMaker* constructs cross-validation folds from a dataset; the *TrainTestSplitMaker* splits it into training and test sets by holding part of the data out for the test set. The *ClassAssigner* allows you to decide which attribute is the class. With *ClassValuePicker* you choose a value that is treated as the *positive* class when generating ROC and other threshold curves. The *ClassifierPerformanceEvaluator* collects evaluation statistics: it can send the textual evaluation to a text viewer and the threshold curves to a performance chart. The *IncrementalClassifierEvaluator* performs the same function for incremental classifiers: it computes running squared errors and so on. There is also a *ClustererPerformanceEvaluator,* which is similar to the *ClassifierPerformanceEvaluator.* The *PredictionAppender* takes a classifier and a dataset and appends the classifier's predictions to the dataset.

11.3 Configuring and connecting the components

You establish the knowledge flow by configuring the individual components and connecting them up. Figure 11.3 shows typical operations that are available by right-clicking the various component types. These menus have up to three sections: *Edit, Connections,* and *Actions.* The *Edit* operations delete components and open up their configuration panel. Classifiers and filters are configured just as in the Explorer. Data sources are configured by opening a file (as we saw previously), and evaluation components are configured by setting parameters such as the number of folds for cross-validation. The *Actions* operations are specific to that type of component, such as starting to load data from a data source or opening a window to show the results of visualization. The *Connections* operations are used to connect components together by selecting the type of connection from the source component and then clicking on the target object. Not

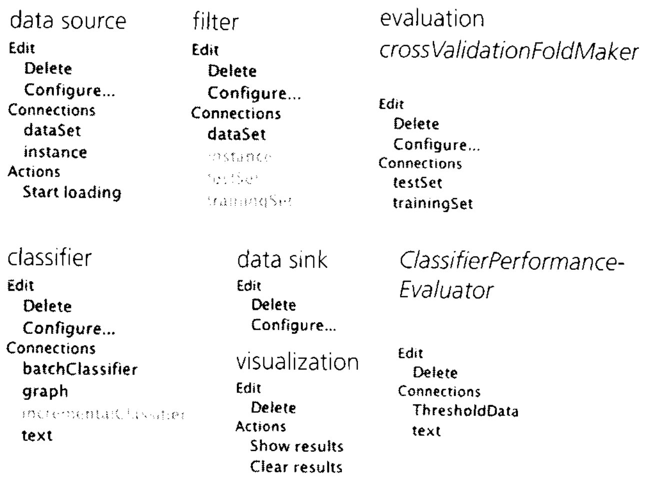


Figure 11.3 Operations on the Knowledge Flow components.

all targets are suitable; applicable ones are highlighted. Items on the connections menu are disabled (grayed out) until the component receives other connections that render them applicable.

There are two kinds of connection from data sources: *dataset* connections and *instance* connections. The former are for batch operations such as classifiers like *J48;* the latter are for stream operations such as *NaiveBayesUpdateable.* A data source component cannot provide both types of connection: once one is selected, the other is disabled. When a *dataset* connection is made to a batch classifier, the classifier needs to know whether it is intended to serve as a training set or a test set. To do this, you first make the data source into a test or training set using the *TestSetMaker* or *TrainingSetMaker* components from the *Evaluation* panel. On the other hand, an *instance* connection to an incremental classifier is made directly: there is no distinction between training and testing because the instances that flow update the classifier incrementally. In this case a prediction is made for each incoming instance and incorporated into the test results; then the classifier is trained on that instance. If you make an *instance* connection to a batch classifier it will be used as a test instance because training cannot possibly be incremental whereas testing always can be. Conversely, it is quite possible to test an incremental classifier in batch mode using a *dataset* connection.

Connections from a filter component are enabled when it receives input from a data source, whereupon follow-on *dataset* or *instance* connections can be made. *Instance* connections cannot be made to supervised filters or to unsupervised filters that cannot handle data incrementally (such as *Discretize*). To get a test or training set out of a filter, you need to put the appropriate kind in.

The classifier menu has two types of connection. The first type, namely, *graph* and *text* connections, provides graphical and textual representations of the classifier's learned state and is only activated when it receives a training set input. The other type, namely, *batchClassifier* and *incrementalClassifier* connections, makes data available to a performance evaluator and is only activated when a test set input is present, too. Which one is activated depends on the type of the classifier.

Evaluation components are a mixed bag. *TrainingSetMaker* and *TestSetMaker* turn a dataset into a training or test set. *CrossValidationFoldMaker* turns a dataset into *both* a training set and a test set. *ClassifierPerformanceEvaluator* (used in the example of Section 11.1) generates textual and graphical output for visualization components. Other evaluation components operate like filters: they enable follow-on *dataset, instance, training set,* or *test set* connections depending on the input (e.g., *ClassAssigner* assigns a class to a dataset).

Visualization components do not have connections, although some have actions such as *Show results* and *Clear results.*

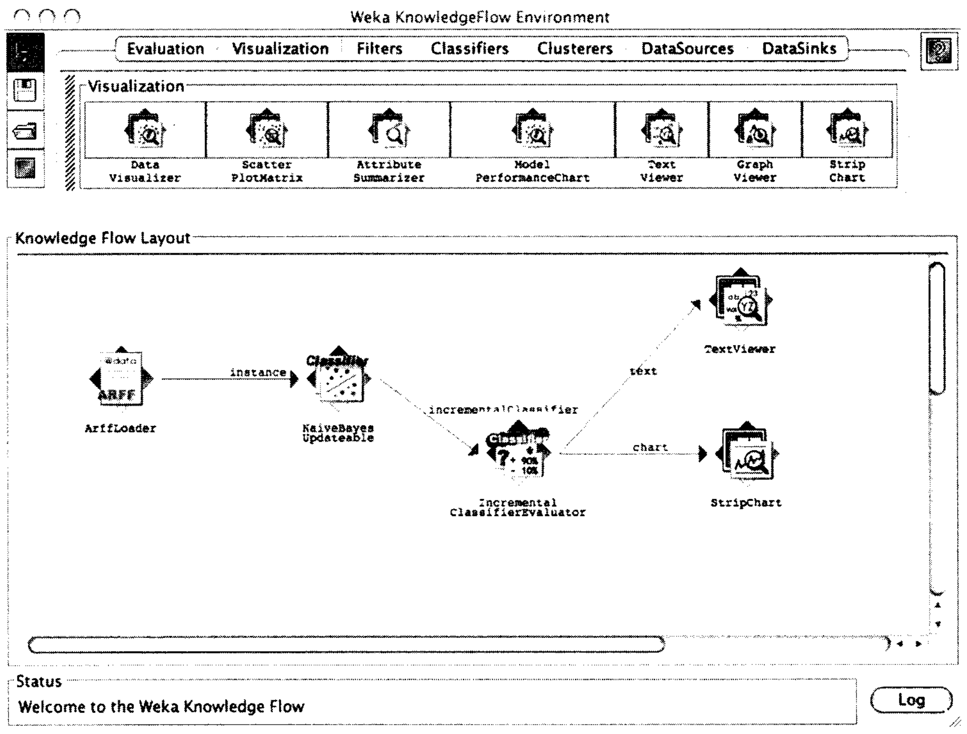
11.4 Incremental learning

In most respects the Knowledge Flow interface is functionally similar to the Explorer: you can do similar things with both. It does provide some additional flexibility—for example, you can see the tree that *J48* makes for each cross-validation fold. But its real strength is the potential for incremental operation.

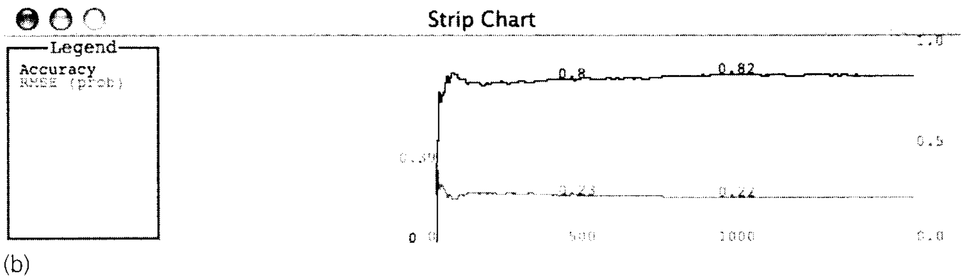
Weka has several classifiers that can handle data incrementally: *AODE,* a version of Naïve Bayes *(NaiveBayesUpdateable), Winnow,* and instance-based learners *(IB1, IBk, KStar, LWL).* The metalearner *RacedIncrementalLogitBoost* operates incrementally (page 416). All filters that work instance by instance are incremental: *Add, AddExpression, Copy, FirstOrder, MakeIndicator, MergeTwoValues, NonSparseToSparse, NumericToBinary, NumericTransform, Obfuscate, Remove, RemoveType, RemoveWithValues, SparseToNonSparse,* and *SwapValues.*

If all components connected up in the Knowledge Flow interface operate incrementally, so does the resulting learning system. It does not read in the dataset before learning starts, as the Explorer does. Instead, the data source component reads the input instance by instance and passes it through the Knowledge Flow chain.

Figure 11.4(a) shows a configuration that works incrementally. An *instance* connection is made from the loader to the updatable Naïve Bayes classifier. The classifier's text output is taken to a viewer that gives a textual description



(a)



(b)

Figure 11.4 A Knowledge Flow that operates incrementally: (a) the configuration and (b) the strip chart output.

of the model. Also, an *incrementalClassifier* connection is made to the corresponding performance evaluator. This produces an output of type *chart*, which is piped to a strip chart visualization component to generate a scrolling data plot.

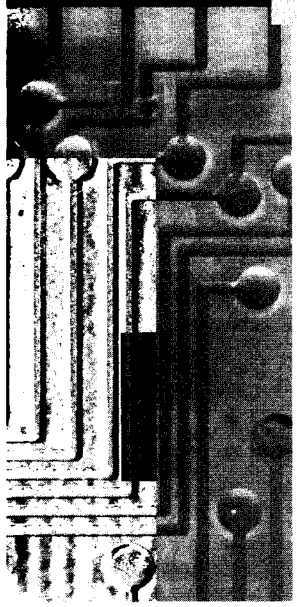
Figure 11.4(b) shows the strip chart output. It plots both the accuracy and the root mean-squared probability error against time. As time passes, the whole plot (including the axes) moves leftward to make room for new data at the right.

When the vertical axis representing time 0 can move left no farther, it stops and the time origin starts to increase from 0 to keep pace with the data coming in at the right. Thus when the chart is full it shows a window of the most recent time units. The strip chart can be configured to alter the number of instances shown on the x axis.

This particular Knowledge Flow configuration can process input files of any size, even ones that do not fit into the computer's main memory. However, it all depends on how the classifier operates internally. For example, although they are incremental, many instance-based learners store the entire dataset internally.

chapter 12

The Experimenter

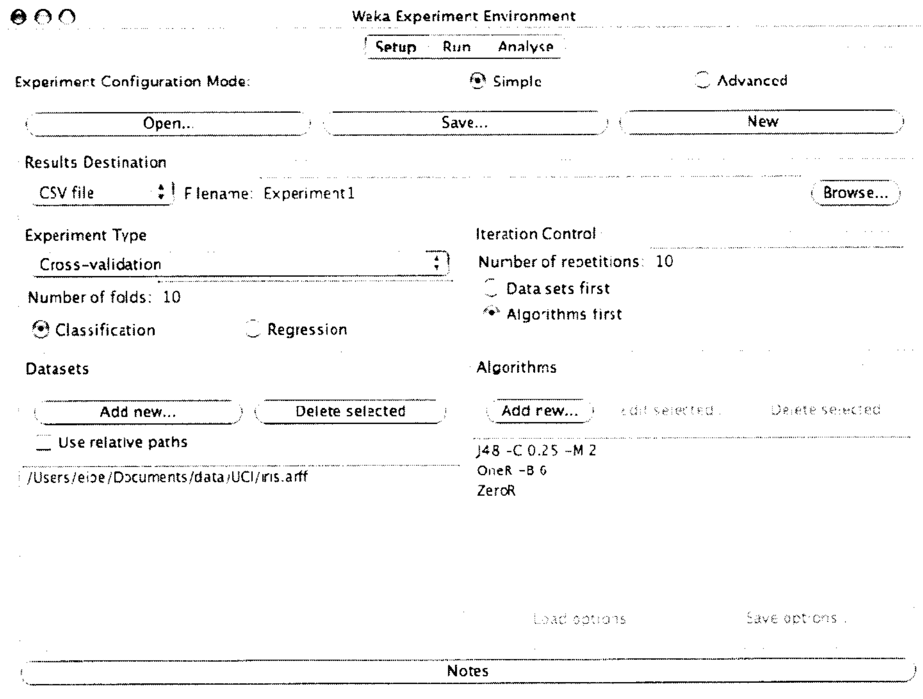


The Explorer and Knowledge Flow environments help you determine how well machine learning schemes perform on given datasets. But serious investigative work involves substantial experiments—typically running several learning schemes on different datasets, often with various parameter settings—and these interfaces are not really suitable for this. The Experimenter enables you to set up large-scale experiments, start them running, leave them, and come back when they have finished and analyze the performance statistics that have been collected. They automate the experimental process. The statistics can be stored in ARFF format, and can themselves be the subject of further data mining. You invoke this interface by selecting *Experimenter* from the choices at the bottom of the panel in Figure 10.3(a).

Whereas the Knowledge Flow transcends limitations of space by allowing machine learning runs that do not load in the whole dataset at once, the Experimenter transcends limitations of time. It contains facilities for advanced Weka users to distribute the computing load across multiple machines using Java RMI. You can set up big experiments and just leave them to run.

12.1 Getting started

As an example, we will compare the J4.8 decision tree method with the baseline methods *OneR* and *ZeroR* on the Iris dataset. The Experimenter has three panels: *Setup, Run,* and *Analyze.* Figure 12.1(a) shows the first: you select the others from the tabs at the top. Here, the experiment has already been set up. To do this, first click *New* (toward the right at the top) to start a new experiment (the other two buttons in that row save an experiment and open a previously saved one). Then, on the line below, select the destination for the results—in this case the file *Experiment1*—and choose *CSV file.* Underneath, select the datasets—we have only one, the iris data. To the right of the datasets, select the algorithms to be tested—we have three. Click *Add new* to get a standard Weka object editor from which you can choose and configure a classifier. Repeat this operation to add the three classifiers. Now the experiment is ready. The other settings shown in Figure 12.1(a) are all default values. If you want to reconfigure a classifier that is already in the list, you can use the *Edit selected* button. You can also save the options for a particular classifier in XML format for later reuse.



Experiment Configuration Mode: [✓ Simple] Advanced

Results Destination
CSV file Filename: Experiment1

Experiment Type
Cross-validation
Number of folds: 10
[✓ Classification] Regression

Iteration Control
Number of repetitions: 10
Data sets first [✓ Algorithms first]

Datasets
Use relative paths
/Users/eibe/Documents/data/UCI/iris.arff

Algorithms
J48 -C 0.25 -M 2
OneR -B 6
ZeroR

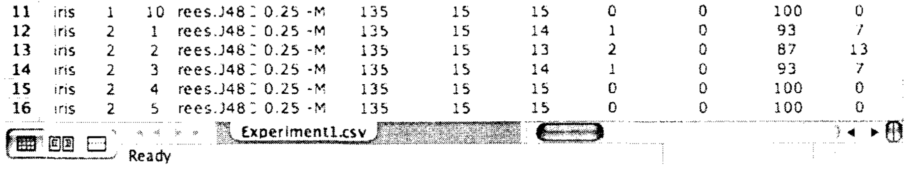
(a)

Figure 12.1 An experiment: (a) setting it up, (b) the results file, and (c) a spreadsheet with the results.

```
Dataset,Run,Fold,Scheme,Scheme_options,Scheme_version_ID,Date_time,Number
_of_training_instances,Number_of_testing_instances,Number_correct,Number_
incorrect,Number_unclassified,Percent_correct,Percent_incorrect,Percent_u
nclassified,Kappa_statistic,Mean_absolute_error,Root_mean_squared_error,R
elative_absolute_error,Root_relative_squared_error,SF_prior_entropy,SF_sc
heme_entropy,SF_entropy_gain,SF_mean_prior_entropy,SF_mean_scheme_entropy
,SF_mean_entropy_gain,KB_information,KB_mean_information,KB_relative_info
rmation,True_positive_rate,Num_true_positives,False_positive_rate,Num_fal
se_positives,True_negative_rate,Num_true_negatives,False_negative_rate,Nu
m_false_negatives,IR_precision,IR_recall,F_measure,Time_training,Time_tes
ting,Summary,measureTreeSize,measureNumLeaves,measureNumRules

iris,1,1,weka.classifiers.trees.J48,'-C 0.25 -M 2',-217733168393644444,2.
00405230549E7,135.0,15.0,14.0,1.0,0.0,93.33333333333333,6.666666666666667
,0.0,0.9,0.0450160137965016,0.1693176548766098,10.128603104212857,35.9176
98581356284,23.77443751081735,2.632715099281766,21.141722411535582,1.5849
625007211567,0.17551433995211774,1.4094481607690388,21.615653599867994,1.
4410435733245328,1363.79589990507,1.0,5.0,0.0,0.0,1.0,10.0,0.0,0.0,1.0,1.
0,1.0,0.0070,0.0,'Number of leaves: 4\nSize of the tree: 7\n',7.0,4.0,4.0
```

(b)



Experiment1.csv

	A	B	C	D	E	F	G	H	I	J	K	L
1	Data-set	Run	Folc	Scheme	Scheme options	Number of train instances	Number of test instances	Number correct	Number incorrect	Number unclassified	Percent correct	Percent incorrect
2	iris	1	1	rees.J48	0.25 -M	135	15	14	1	0	93	7
3	iris	1	2	rees.J48	0.25 -M	135	15	15	0	0	100	0
4	iris	1	3	rees.J48	0.25 -M	135	15	15	0	0	100	0
5	iris	1	4	rees.J48	0.25 -M	135	15	15	0	0	100	0
6	iris	1	5	rees.J48	0.25 -M	135	15	14	1	0	93	7
7	iris	1	6	rees.J48	0.25 -M	135	15	15	0	0	100	0
8	iris	1	7	rees.J48	0.25 -M	135	15	13	2	0	87	13
9	iris	1	8	rees.J48	0.25 -M	135	15	13	2	0	87	13
10	iris	1	9	rees.J48	0.25 -M	135	15	15	0	0	100	0
11	iris	1	10	rees.J48	0.25 -M	135	15	15	0	0	100	0
12	iris	2	1	rees.J48	0.25 -M	135	15	14	1	0	93	7
13	iris	2	2	rees.J48	0.25 -M	135	15	13	2	0	87	13
14	iris	2	3	rees.J48	0.25 -M	135	15	14	1	0	93	7
15	iris	2	4	rees.J48	0.25 -M	135	15	15	0	0	100	0
16	iris	2	5	rees.J48	0.25 -M	135	15	15	0	0	100	0

(c)

Figure 12.1 (continued)

Running an experiment

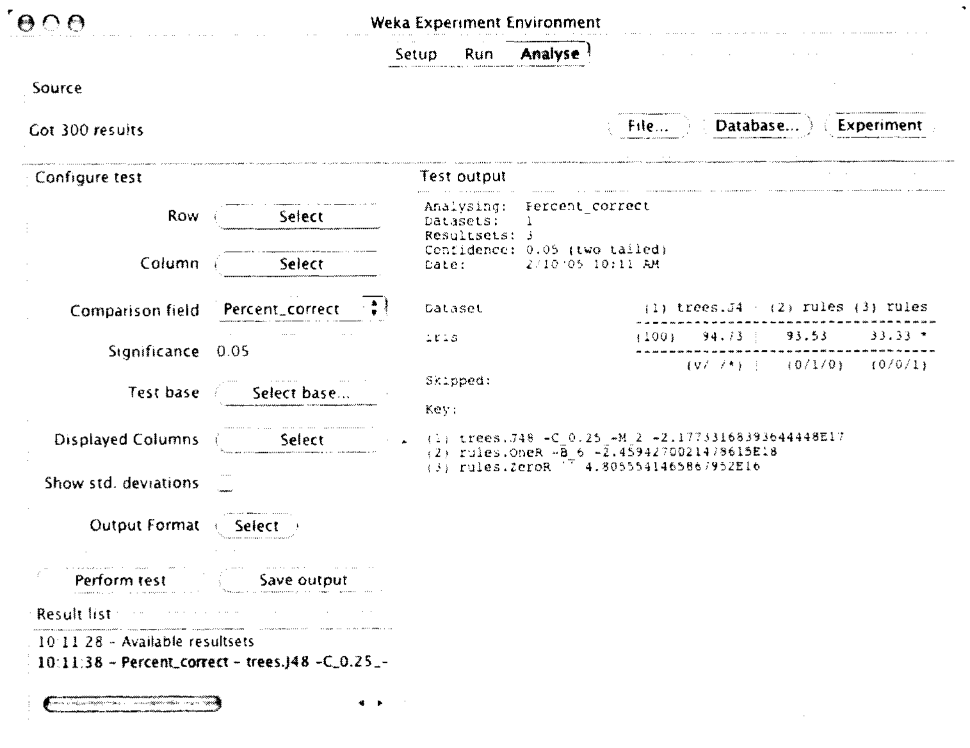
To run the experiment, click the *Run* tab, which brings up a panel that contains a *Start* button (and little else); click it. A brief report is displayed when the operation is finished. The file *Experiment1.csv* contains the results. The first two lines are shown in Figure 12.1(b): they are in CSV format and can be read directly into a spreadsheet, the first part of which appears in Figure 12.1(c). Each row represents 1 fold of a 10-fold cross-validation (see the *Fold* column). The cross-validation is run 10 times (the *Run* column) for each classifier (the *Scheme* column). Thus the file contains 100 rows for each classifier, which makes 300 rows in all (plus the header row). Each row contains plenty of information—46 columns, in fact—including the options supplied to the machine learning scheme; the number of training and test instances; the number (and percentage) of correct, incorrect, and unclassified instances; the mean absolute error, root mean-squared error, and many more.

There is a great deal of information in the spreadsheet, but it is hard to digest. In particular, it is not easy to answer the question posed previously: how does J4.8 compare with the baseline methods *OneR* and *ZeroR* on this dataset? For that we need the *Analyze* panel.

Analyzing the results

The reason that we generated the output in CSV format was to show the spreadsheet in Figure 12.1(c). The Experimenter normally produces its output in ARFF format. You can also leave the file name blank, in which case the Experimenter stores the results in a temporary file.

The *Analyze* panel is shown in Figure 12.2. To analyze the experiment that has just been performed, click the *Experiment* button at the right near the top; otherwise, supply a file that contains the results of another experiment. Then click *Perform test* (near the bottom on the left). The result of a statistical signifi-



Got 300 results

Comparison field: Percent_correct

Significance: 0.05

Show std. deviations

10:11:28 - Available resultsets
10:11:38 - Percent_correct - trees.J48 -C_0.25_-

```
Analysing:  Percent_correct
Datasets:   1
Resultsets: 3
Confidence: 0.05 (two tailed)
Date:       2/10/05 10:11 AM

Dataset                 (1) trees.J4 | (2) rules (3) rules
                        ----------------------------------
iris                    (100)  94.73 |   93.53     33.33 *
                        ----------------------------------
                              (v/ /*) |  (0/1/0)   (0/0/1)

Skipped:

Key:

(1) trees.J48 -C_0.25_-M_2 -2.17733168393644448E17
(2) rules.OneR -B_6 -2.459427002147861E18
(3) rules.ZeroR '' 4.805554146586 7952E16
```

Figure 12.2 Statistical test results for the experiment in Figure 12.1.

cance test of the performance of the first learning scheme *(J48)* versus that of the other two (*OneR* and *ZeroR*) will be displayed in the large panel on the right.

We are comparing the percent correct statistic: this is selected by default as the comparison field shown toward the left of Figure 12.2. The three methods are displayed horizontally, numbered *(1)*, *(2)*, and *(3)*, as the heading of a little table. The labels for the columns are repeated at the bottom—*trees.J48*, *rules.OneR*, and *rules.ZeroR*—in case there is insufficient space for them in the heading. The inscrutable integers beside the scheme names identify which version of the scheme is being used. They are present by default to avoid confusion among results generated using different versions of the algorithms. The value in parentheses at the beginning of the *iris* row *(100)* is the number of experimental runs: 10 times 10-fold cross-validation.

The percentage correct for the three schemes is shown in Figure 12.2: 94.73% for method 1, 93.53% for method 2, and 33.33% for method 3. The symbol placed beside a result indicates that it is statistically better *(v)* or worse *(*)* than the baseline scheme—in this case J4.8—at the specified significance level (0.05, or 5%). The corrected resampled *t*-test from Section 5.5 (page 157) is used. Here, method 3 is significantly worse than method 1, because its success rate is followed by an asterisk. At the bottom of columns 2 and 3 are counts (x/y/z) of the number of times the scheme was better than (x), the same as (y), or worse than (z) the baseline scheme on the datasets used in the experiment. In this case there is only one dataset; method 2 was equivalent to method 1 (the baseline) once, and method 3 was worse than it once. (The annotation *(v/ /*)* is placed at the bottom of column 1 to help you remember the meanings of the three counts x/y/z.)

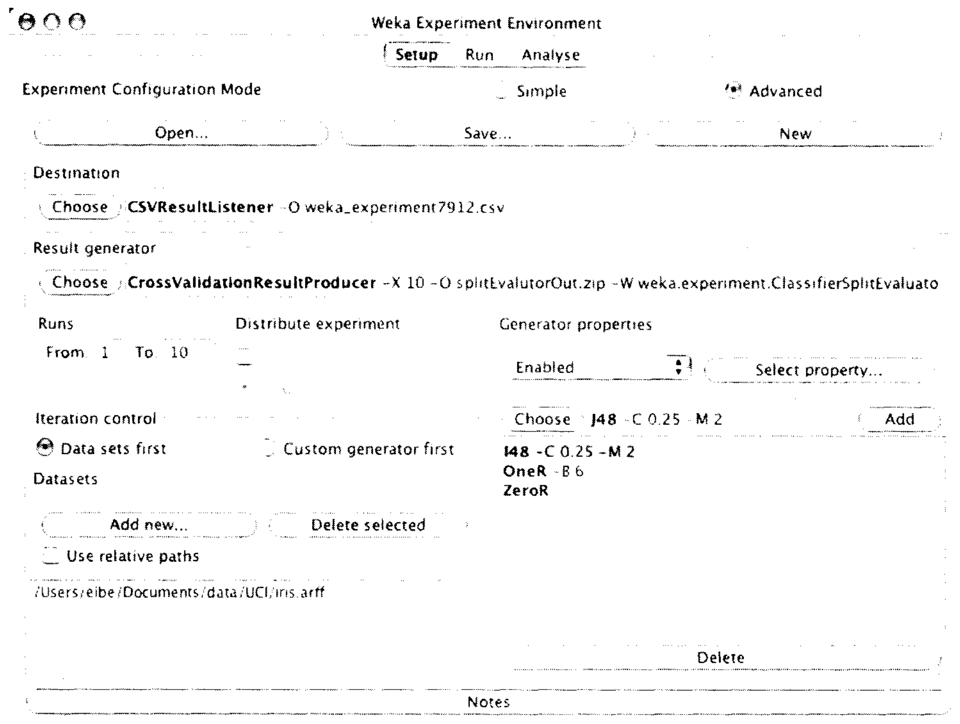
12.2 Simple setup

In the *Setup* panel shown in Figure 12.1(a) we left most options at their default values. The experiment is a 10-fold cross-validation repeated 10 times. You can alter the number of folds in the box at center left and the number of repetitions in the box at center right. The experiment type is classification; you can specify regression instead. You can choose several datasets, in which case each algorithm is applied to each dataset, and change the order of iteration using the *Data sets first* and *Algorithm first* buttons. The alternative to cross-validation is the holdout method. There are two variants, depending on whether the order of the dataset is preserved or the data is randomized. You can specify the percentage split (the default is two-thirds training set and one-third test set).

Experimental setups can be saved and reopened. You can make notes about the setup by pressing the *Notes* button, which brings up an editor window. Serious Weka users soon find the need to open up an experiment and rerun it with some modifications—perhaps with a new dataset or a new learning algorithm. It would be nice to avoid having to recalculate all the results that have already been obtained! If the results have been placed in a database rather than an ARFF or CSV file, this is exactly what happens. You can choose *JDBC database* in the results destination selector and connect to any database that has a JDBC driver. You need to specify the database's URL and enter a username and password. To make this work with your database you may need to modify the *weka/experiment/DatabaseUtils.props* file in the Weka distribution. If you alter an experiment that uses a database, Weka will reuse previously computed results whenever they are available. This greatly simplifies the kind of iterative experimentation that typically characterizes data mining research.

12.3 Advanced setup

The Experimenter has an advanced mode. Click near the top of the panel shown in Figure 12.1(a) to obtain the more formidable version of the panel shown in Figure 12.3. This enlarges the options available for controlling the experiment—including, for example, the ability to generate learning curves. However, the



Experiment Configuration Mode: Simple [✓ Advanced]

Destination
CSVResultListener -O weka_experiment7912.csv

Result generator
CrossValidationResultProducer -X 10 -O splitEvalutorOut.zip -W weka.experiment.ClassifierSplitEvaluato

Runs
From 1 To 10

Distribute experiment

Generator properties
Enabled

J48 -C 0.25 -M 2

Iteration control
[✓ Data sets first] Custom generator first

Datasets
Use relative paths
/Users/eibe/Documents/data/UCI/iris.arff

J48 -C 0.25 -M 2
OneR -B 6
ZeroR

Figure 12.3 Setting up an experiment in advanced mode.

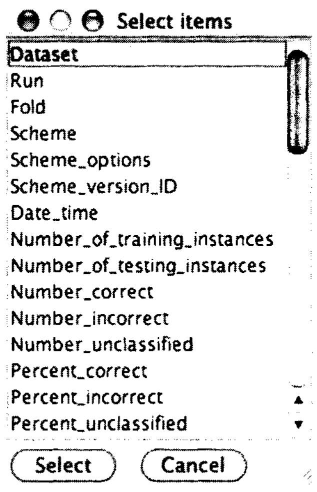
advanced mode is hard to use, and the simple version suffices for most purposes. For example, in advanced mode you can set up an iteration to test an algorithm with a succession of different parameter values, but the same effect can be achieved in simple mode by putting the algorithm into the list several times with different parameter values. Something you may need the advanced mode for is to set up distributed experiments, which we describe in Section 12.5.

12.4 The Analyze panel

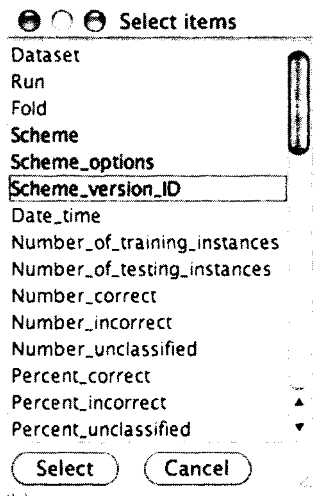
Our walkthrough used the *Analyze* panel to perform a statistical significance test of one learning scheme *(J48)* versus two others (*OneR* and *ZeroR*). The test was on the error rate—the *Comparison* field in Figure 12.2. Other statistics can be selected from the drop-down menu instead: percentage incorrect, percentage unclassified, root mean-squared error, the remaining error measures from Table 5.8 (page 178), and various entropy figures. Moreover, you can see the standard deviation of the attribute being evaluated by ticking the *Show std deviations* checkbox.

Use the *Select base* menu to change the baseline scheme from J4.8 to one of the other learning schemes. For example, selecting *OneR* causes the others to be compared with this scheme. In fact, that would show that there is a statistically significant difference between *OneR* and *ZeroR* but not between *OneR* and *J48.* Apart from the learning schemes, there are two other choices in the *Select base* menu: *Summary* and *Ranking.* The former compares each learning scheme with every other scheme and prints a matrix whose cells contain the number of datasets on which one is significantly better than the other. The latter ranks the schemes according to the total number of datasets that represent wins (>) and losses (<) and prints a league table. The first column in the output gives the difference between the number of wins and the number of losses.

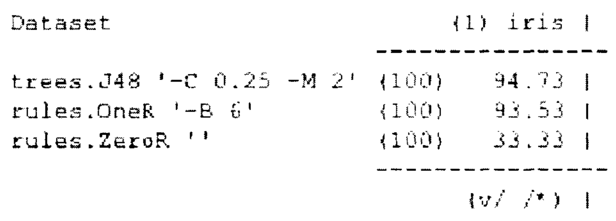
The *Row* and *Column* fields determine the dimensions of the comparison matrix. Clicking *Select* brings up a list of all the features that have been measured in the experiment—in other words, the column labels of the spreadsheet in Figure 12.1(c). You can select which to use as the rows and columns of the matrix. (The selection does not appear in the *Select* box because more than one parameter can be chosen simultaneously.) Figure 12.4 shows which items are selected for the rows and columns of Figure 12.2. The two lists show the experimental parameters (the columns of the spreadsheet). *Dataset* is selected for the rows (and there is only one in this case, the Iris dataset), and *Scheme, Scheme options,* and *Scheme_version_ID* are selected for the column (the usual convention of shift-clicking selects multiple entries). All three can be seen in Figure 12.2—in fact, they are more easily legible in the key at the bottom.



(a)

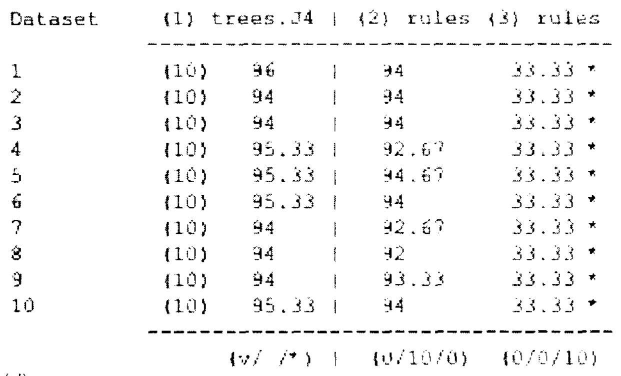


(b)



```
Dataset                          (1) iris  |
                                ---------------
trees.J48 '-C 0.25 -M 2' (100)    94.73 |
rules.OneR '-B 6'         (100)    93.53 |
rules.ZeroR ''            (100)    33.33 |
                                ---------------
                                   (v/ /*) |
```

(c)



```
Dataset     (1) trees.J4 | (2) rules (3) rules
            -----------------------------------
1           (10)  96     |  94          33.33 *
2           (10)  94     |  94          33.33 *
3           (10)  94     |  94          33.33 *
4           (10)  95.33  |  92.67       33.33 *
5           (10)  95.33  |  94.67       33.33 *
6           (10)  95.33  |  94          33.33 *
7           (10)  94     |  92.67       33.33 *
8           (10)  94     |  92          33.33 *
9           (10)  94     |  93.33       33.33 *
10          (10)  95.33  |  94          33.33 *
            -----------------------------------
                 (v/ /*) | (0/10/0)   (0/0/10)
```

(d)

Figure 12.4 Rows and columns of Figure 12.2: (a) row field, (b) column field, (c) result of swapping the row and column selections, and (d) substituting *Run* for *Dataset* as rows.

If the row and column selections were swapped and the *Perform test* button were pressed again, the matrix would be transposed, giving the result in Figure 12.4(c). There are now three rows, one for each algorithm, and one column, for the single dataset. If instead the row of *Dataset* were replaced by *Run* and the test were performed again, the result would be as in Figure 12.4(d). *Run* refers to the runs of the cross-validation, of which there are 10, so there are now 10 rows. The number in parentheses after each row label (100 in Figure 12.4(c) and 10 in Figure 12.4(d)) is the number of results corresponding to that row—in other words, the number of measurements that participate in the averages displayed by the cells in that row. There is also a button that allows you to select a subset of columns to display (the baseline column is always included), and another that allows you to select the output format: plain text (default), output for the LaTeX typesetting system, and CSV format.

12.5 Distributing processing over several machines

A remarkable feature of the Experimenter is that it can split up an experiment and distribute it across several processors. This is for advanced Weka users and is only available from the advanced version of the *Setup* panel. Some users avoid working with this panel by setting the experiment up on the simple version and switching to the advanced version to distribute it, because the experiment's structure is preserved when you switch. However, distributing an experiment *is* an advanced feature and is often difficult. For example, file and directory permissions can be tricky to set up.

Distributing an experiment works best when the results are all sent to a central database by selecting *JDBC database* as the results destination in the panel shown in Figure 12.1(a). It uses the RMI facility, and works with any database that has a JDBC driver. It has been tested on several freely available databases. Alternatively, you could instruct each host to save its results to a different ARFF file and merge the files afterwards.

To distribute an experiment, each host must (1) have Java installed, (2) have access to whatever datasets you are using, and (3) be running the *weka.experiment.RemoteEngine* experiment server. If results are sent to a central database, the appropriate JDBC drivers must be installed on each host. Getting all this right is the difficult part of running distributed experiments.

To initiate a remote engine experiment server on a host machine, first copy *remoteExperimentServer.jar* from the Weka distribution to a directory on the host. Unpack it with

```
jar xvf remoteExperimentServer.jar
```

It expands to two files: *remoteEngine.jar*, an executable *jar* file that contains the experiment server, and *remote.policy*.

The *remote.policy* file grants the remote engine permission to perform certain operations, such as connecting to ports or accessing a directory. It needs to be edited to specify correct paths in some of the permissions; this is self-explanatory when you examine the file. By default, it specifies that code can be downloaded on HTTP port 80 from anywhere on the Web, but the remote engines can also load code from a file URL instead. To arrange this, uncomment the example and replace the pathname appropriately. The remote engines also need to be able to access the datasets used in an experiment (see the first entry in *remote.policy*). The paths to the datasets are specified in the Experimenter (i.e., the client), and the same paths must be applicable in the context of the remote engines. To facilitate this it may be necessary to specify relative pathnames by selecting the *Use relative paths* tick box shown in the *Setup* panel of the Experimenter.

To start the remote engine server, type

```
java -classpath remoteEngine.jar:<path_to_any_jdbc_drivers>
     -Djava.security.policy=remote.policy weka.experiment.RemoteEngine
```

from the directory containing *remoteEngine.jar*. If all goes well you will see this message (or something like it):

```
Host name : ml.cs.waikato.ac.nz
RemoteEngine exception: Connection refused to host:
ml.cs.waikato.ac.nz; nested exception is:
     java.net.ConnectException: Connection refused
Attempting to start rmi registry...
RemoteEngine bound in RMI registry
```

Despite initial appearances, this is good news! The connection was refused because no RMI registry was running on that server, and hence the remote engine has started one. Repeat the process on all hosts. It does not make sense to run more than one remote engine on a machine.

Start the Experimenter by typing

```
java -Djava.rmi.server.codebase=<URL_for_weka_code>
     weka.gui.experiment.Experimenter
```

The URL specifies where the remote engines can find the code to be executed. If the URL denotes a directory (i.e., one that contains the Weka directory) rather than a *jar* file, it must end with path separator (e.g., /).

The Experimenter's advanced *Setup* panel in Figure 12.3 contains a small pane at center left that determines whether an experiment will be distributed or not. This is normally inactive. To distribute the experiment click the checkbox to activate the *Hosts* button; a window will pop up asking for the machines over which to distribute the experiment. Host names should be fully qualified (e.g., *ml.cs.waikato.ac.nz*).

Having entered the hosts, configure the rest of the experiment in the usual way (better still, configure it before switching to the advanced setup mode). When the experiment is started using the *Run* panel, the progress of the subexperiments on the various hosts is displayed, along with any error messages.

Distributing an experiment involves splitting it into subexperiments that RMI sends to the hosts for execution. By default, experiments are partitioned by dataset, in which case there can be no more hosts than there are datasets. Then each subexperiment is self-contained: it applies all schemes to a single dataset. An experiment with only a few datasets can be partitioned by run instead. For example, a 10 times 10-fold cross-validation would be split into 10 subexperiments, 1 per run.