**The WEKA Workbench**

**Eibe Frank, Mark A. Hall, and Ian H. Witten**

**Online Appendix for “Data Mining: Practical Machine Learning Tools and Techniques” Morgan Kaufmann, Fourth Edition, 2016**

2

**Contents**

**1 Introduction to Weka 7** 1.1 What’s in WEKA? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7 1.2 How do you use it? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8 1.3 What else can you do? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8 1.4 How do you get it? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9 1.5 The package management system . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

**2 The Explorer 15** 2.1 Getting started . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16 2.1.1 Preparing the data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17 2.1.2 Loading the data into the Explorer . . . . . . . . . . . . . . . . . . . . . . 17 2.1.3 Building a decision tree . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18 2.1.4 Examining the output . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19 2.1.5 Doing it again . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19 2.1.6 Working with models . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21 2.1.7 When things go wrong . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22 2.2 Exploring the Explorer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22 2.2.1 Loading and filtering files . . . . . . . . . . . . . . . . . . . . . . . . . . . 23 2.2.2 Converting files to ARFF . . . . . . . . . . . . . . . . . . . . . . . . . . . 23 2.2.3 Using filters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24 2.2.4 Training and testing learning schemes . . . . . . . . . . . . . . . . . . . . 25 2.2.5 Using a metalearner . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26 2.2.6 Clustering and association rules . . . . . . . . . . . . . . . . . . . . . . . . 26 2.2.7 Attribute selection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27 2.2.8 Visualization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27 2.3 Filtering algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27 2.3.1 Unsupervised attribute filters . . . . . . . . . . . . . . . . . . . . . . . . . 28 2.3.1.1 Adding and removing attributes . . . . . . . . . . . . . . . . . . 28 2.3.1.2 Changing values . . . . . . . . . . . . . . . . . . . . . . . . . . . 30 2.3.1.3 Conversions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30 2.3.1.4 String conversion . . . . . . . . . . . . . . . . . . . . . . . . . . . 31 2.3.1.5 Time series . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32 2.3.1.6 Randomizing the attributes . . . . . . . . . . . . . . . . . . . . . 32 2.3.2 Unsupervised instance filters . . . . . . . . . . . . . . . . . . . . . . . . . 32 2.3.2.1 Randomizing and subsampling . . . . . . . . . . . . . . . . . . . 32 2.3.2.2 Sparse instances . . . . . . . . . . . . . . . . . . . . . . . . . . . 33 2.3.3 Supervised filters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

3

4 CONTENTS

2.3.3.1 Supervised attribute filters . . . . . . . . . . . . . . . . . . . . . 34 2.3.3.2 Supervised instance filters . . . . . . . . . . . . . . . . . . . . . . 34 2.4 Learning algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35 2.4.1 Bayesian classifiers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35 2.4.2 Trees . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36 2.4.3 Rules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37 2.4.4 Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37 2.4.5 Neural networks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39 2.4.6 Lazy classifiers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40 2.4.7 Miscellaneous classifiers . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40 2.4.8 Metalearning algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41 2.4.8.1 Bagging and randomization . . . . . . . . . . . . . . . . . . . . . 41 2.4.8.2 Boosting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41 2.4.8.3 Combining classifiers . . . . . . . . . . . . . . . . . . . . . . . . 42 2.4.8.4 Cost-sensitive learning . . . . . . . . . . . . . . . . . . . . . . . . 42 2.4.8.5 Optimizing performance . . . . . . . . . . . . . . . . . . . . . . . 42 2.4.8.6 Retargeting classifiers for different tasks . . . . . . . . . . . . . . 43 2.5 Clustering algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43 2.6 Association rule learners . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44 2.7 Attribute selection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45 2.7.1 Attribute subset evaluators . . . . . . . . . . . . . . . . . . . . . . . . . . 45 2.7.2 Single-attribute evaluators . . . . . . . . . . . . . . . . . . . . . . . . . . . 46 2.7.3 Search methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

**3 The Knowledge Flow Interface 73** 3.1 Getting started . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73 3.2 Knowledge Flow components . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75 3.3 Configuring and connecting the components . . . . . . . . . . . . . . . . . . . . . 77 3.4 Incremental learning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

**4 The Experimenter 81** 4.1 Getting started . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81 4.2 Running an experiment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83 4.3 Analyzing the results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83 4.4 Simple setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84 4.5 Advanced setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85 4.6 The Analyze panel . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87 4.7 Distributing processing over several machines . . . . . . . . . . . . . . . . . . . . 89

**5 The Command-Line Interface 91** 5.1 Getting started . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91 5.1.1 weka.Run . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91 5.2 The structure of WEKA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92 5.2.1 Classes, instances, and packages . . . . . . . . . . . . . . . . . . . . . . . 92 5.2.2 The weka.core package . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93 5.2.3 The weka.classifiers package . . . . . . . . . . . . . . . . . . . . . . . . . . 93 5.2.4 Other packages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102 5.2.5 Javadoc indexes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102 5.3 Command-line options . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102

CONTENTS 5

5.3.1 Generic options . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104 5.3.2 Scheme-specific options . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105

**6 Embedded Machine Learning 107** 6.1 A simple data mining application . . . . . . . . . . . . . . . . . . . . . . . . . . . 107 6.2 main() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107 6.3 messageClassifier() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111 6.4 updateData() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111 6.5 classifyMessage() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112

**7 Writing New Learning Schemes 113** 7.1 An example classifier . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113 7.1.1 buildClassifier() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 120 7.1.2 makeTree() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 120 7.1.3 computeInfoGain() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121 7.1.4 classifyInstance() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121 7.1.5 toSource() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122 7.1.6 main() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122 7.2 Conventions for implementing classifiers . . . . . . . . . . . . . . . . . . . . . . . 126 7.2.1 Capabilities . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 126

6 CONTENTS

**Chapter 1**

**Introduction to Weka**

The WEKA workbench is a collection of machine learning algorithms and data preprocessing tools that includes virtually all the algorithms described in our book. It is designed so that you can quickly try out existing methods on new datasets in flexible ways. It provides extensive support for the whole process of experimental data mining, including preparing the input data, evaluating learning schemes statistically, and visualizing the input data and the result of learning. As well as a wide variety of learning algorithms, it includes a wide range of preprocessing tools. This diverse and comprehensive toolkit is accessed through a common interface so that its users can compare different methods and identify those that are most appropriate for the problem at hand. WEKA was developed at the University of Waikato in New Zealand; the name stands for *Waikato Environment for Knowledge Analysis*. Outside the university 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 and distributed under the terms of the GNU General Public License. It runs on almost any platform and has been tested under Linux, Windows, and Macintosh operating systems.

**1.1 What’s in WEKA?**

WEKA provides implementations of learning algorithms that you can easily apply to your dataset. It also includes a variety of tools for transforming datasets, such as the algorithms for discretization and sampling. 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.The workbench includes methods for the main data mining problems: regression, classifica- tion, clustering, association rule mining, and attribute selection. Getting to know the data is an integral part of the work, and many data visualization facilities and data preprocessing tools are provided. All algorithms take their input in the form of a single relational table that can be read from a file or generated by a database query.

One way of using WEKA is to apply a learning method to a dataset and analyze its output to learn more about the data. Another is to use learned models to generate predictions on new instances. A third is to apply several different learners and compare their performance in order to choose one for prediction. In the interactive WEKA interface you select the learning method you want from a menu. Many methods have tunable parameters, which you access through a property sheet or *object editor*. A common evaluation module is used to measure the performance of all classifiers. Implementations of actual learning schemes are the most valuable resource that

7

8 CHAPTER 1. INTRODUCTION TO WEKA

WEKA provides. But tools for preprocessing the data, called *filters*, come a close second. Like classifiers, you select filters from a menu and tailor them to your requirements.

**1.2 How do you use it?**

The easiest way to use WEKA is through a graphical user interface called the *Explorer*. This gives access to all of its facilities using menu selection and form filling. For example, you can quickly read in a dataset from a file and build a decision tree from it. The Explorer guides you by presenting options as forms to be filled out. Helpful *tool tips* pop up as the mouse passes over items on the screen to explain what they do. Sensible default values ensure that you can get results with a minimum of effort—but you will have to think about what you are doing to understand what the results mean.

There are three other graphical user interfaces to WEKA. The *Knowledge Flow* interface allows you to design configurations for streamed data processing. A fundamental disadvantage of the Explorer is that it holds everything in main memory—when you open a dataset, it im- mediately loads it all in. That means that it can only be applied to small-to medium-sized problems. However, WEKA contains some incremental algorithms that can be used to process very large datasets. The Knowledge Flow interface lets you drag boxes representing learning algorithms and data sources around the screen and join them together into the configuration you want. It enables you to specify a data stream by connecting components representing data sources, preprocessing tools, learning algorithms, evaluation methods, and visualization modules. If the filters and learning algorithms are capable of incremental learning, data will be loaded and processed incrementally.

WEKA’s third interface, the *Experimenter*, is designed to help you answer a basic practical question when applying classification and regression techniques: Which methods and parameter values work best for the given problem? There is usually no way to answer this question a priori, and one reason we developed the workbench was to provide an environment that enables WEKA users to compare a variety of learning techniques. This can be done interactively using the Explorer. However, the Experimenter allows you to automate the process by making it easy to run classifiers and filters with different parameter settings on a corpus of datasets, collect performance statistics, and perform significance tests. Advanced users can employ the Exper- imenter to distribute the computing load across multiple machines using Java remote method invocation. In this way you can set up large-scale statistical experiments and leave them to run. The fourth interface, called the Workbench, is a unified graphical user interface that combines the other three (and any plugins that the user has installed) into one application. The Workbench is highly configurable, allowing the user to specify which applications and plugins will appear, along with settings relating to them.

Behind these interactive interfaces lies the basic functionality of WEKA. This can be accessed in raw form by entering textual commands, which gives access to all features of the system. When you fire up WEKA you have to choose among five different user interfaces via the WEKA GUI Chooser: the Explorer, Knowledge Flow, Experimenter, Workbench, and command-line interfaces. Most people choose the Explorer, at least initially.

**1.3 What else can you do?**

An important resource when working with WEKA 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. We will also identify WEKA’s major building blocks, highlighting

1.4. HOW DO YOU GET IT? 9

which parts contain supervised learning methods, which contain tools for data preprocessing, and which contain methods for other learning schemes. The online documentation gives the only complete list of available algorithms because 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 write and test learning schemes of your own.

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 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 will 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. To make full use of it, you must become acquainted with the basic data structures. To help you reach this point, we will describe these structures in more detail and explain an illustrative implementation of a classifier.

**1.4 How do you get it?**

WEKA is available from http://www.cs.waikato.ac.nz/ml/weka. You can download either a platform-specific installer or an executable Java jar file that you run in the usual way if Java is installed. We recommend that you download and install it now, and follow through the examples in the upcoming sections.

**1.5 The package management system**

The WEKA software has evolved considerably since the third edition of this book was published. Many new algorithms and features have been added to the system, a number of which have been contributed by the community. With so many algorithms on offer we felt that the software could be considered overwhelming to the new user. Therefore a number of algorithms and community contributions were removed and placed into plugin *packages*. A package management system was added that allows the user to browse for, and selectively install, packages of interest.

Another motivation for introducing the package management system was to make the process of contributing to the WEKA software easier, and to ease the maintenance burden on the WEKA development team. A contributor of a plugin package is responsible for maintaining its code and hosting the installable archive, while WEKA simply tracks the package metadata. The package system also opens the door to the use of third-party libraries, something that we would have discouraged in the past in order to keep a lightweight footprint for WEKA.

Figure 1.1 shows the main window of the graphical package manger, which can be accessed from the Tools menu in the *GUI Chooser* panel shown in Figure 1.2. The very first time the package manager is accessed it will download information about the currently available packages. This requires an internet connection, however, once the package metadata has been downloaded it is possible to use the package manager to browse package information while offline. Of course, an Internet connection is still required to be able to actually install a package.

10 CHAPTER 1. INTRODUCTION TO WEKA

Figure 1.2: The GUI Chooser.

The package manager presents a list of packages near the top of its window and a panel at the bottom that displays information on the currently selected package in the list. The user can choose to display packages that are available but not yet installed, only packages that are installed, or all packages. The list presents the name of each package, the broad category that it belongs to, the version currently installed (if any), the most recent version of the package available that is compatible with the version of WEKA being used, and a field that, for installed packages, indicates whether the package has been loaded successfully by WEKA or not. Although

Figure 1.1: The graphical package manager.

1.5. THE PACKAGE MANAGEMENT SYSTEM 11

Figure 1.3: Choosing a version of the *DMNBtext* package to install.

not obvious at first glance, it is possible to install older versions of a particular package. The Repository version field in the list is actually a drop-down box. Figure 1.3 shows selecting a version of the *DMNBtext* package to install. The list of packages can be sorted, in ascending or descending order, by clicking on either the package or category column header.

(b) Dependency.

(a) Primary package.

Figure 1.5: Installing a package with dependencies.

The information panel at the bottom of the window has clickable links for each version of a given package. “Latest” always refers to the latest version of the package, and is the same as the highest version number available. Clicking one of these links displays further information, such

Figure 1.4: Additional information for the *DMNBtext* package.

12 CHAPTER 1. INTRODUCTION TO WEKA

as the author of the package, its license, where the installable archive is located, and its depen- dencies. Figure 1.4 shows this information for the *DMNBtext* package. The information about each package is also browsable at the Web location where WEKA’s package metadata is hosted. At the time of writing this can be found at http://weka.sourceforge.net/packageMetaData/. All packages have at least one dependency listed—the minimum version of the core WEKA system that they can work with. Some packages list further dependencies on other packages. For example, the *multiInstanceLearning* package depends on the *multiInstanceFilters* package. When installing *multiInstanceLearning*, and assuming that *multiInstanceFilters* is not already installed, the system will inform the user the *multiInstanceFilters* is required and will be installed automatically. Figure 1.5 shows the dialogs displayed by the package manager when installing *multiInstanceLearning*.

The package manager displays what are known as *official* packages for WEKA. These are packages that have been submitted to the WEKA team for a cursory review and have had their metadata added to the official central metadata repository. For one reason or another, an author of a package might decide to make it available in an unofficial capacity. These packages do not appear in the official list on the web, or in the list displayed by the graphical package manager. If the user knows the URL to an archive containing an unofficial package, it can be installed by using the button in the upper right-hand corner of the package manager window.

Figure 1.6: New/updated packages available.

Whenever a new package, or new version of an existing one, becomes available the package manager informs the user by displaying a large yellow warning icon. Hovering over this icon displays a tool-tip popup that lists the new packages and prompts the user to click the *Refresh repository cache* button. Clicking this button downloads a fresh copy of all the package infor- mation to the user’s computer. Figure 1.6 shows what is displayed in the situation where there is one new package available and one upgrade to an existing package.

Figure 1.7: Changing the load status of a package.

The *Install* and *Uninstall* buttons at the top of the package manager’s window do exactly as their names suggest. More than one package can be installed or uninstalled in one go by selecting

1.5. THE PACKAGE MANAGEMENT SYSTEM 13

multiple entries in the list. By default, WEKA attempts to load all installed packages, and if a package cannot be loaded for some reason a message will be displayed in the *Loaded* column of the list. The user can opt to prevent a particular package from being loaded by selecting it and then clicking the *Toggle load* button. This will mark the package as one that should not be loaded the next time that WEKA is started. This can be useful if an unstable package is generating errors, conflicting with another package (perhaps due to third-party libraries), or otherwise preventing WEKA from operating properly. Figure 1.7 shows what is displayed when a package’s load status is toggled.

14 CHAPTER 1. INTRODUCTION TO WEKA

**Chapter 2**

**The Explorer**

Figure 2.1: The Explorer interface.

WEKA’s main graphical user interface, the Explorer, gives access to all its facilities using menu selection and form filling. It is illustrated in Figure 2.1. To begin, there are six different panels, selected by the tabs at the top, corresponding to the various data mining tasks that WEKA supports. Further panels can become available by installing appropriate packages.

15

16 CHAPTER 2. THE EXPLORER

**2.1 Getting started**

Suppose you have some data and you want to build a decision tree from it. First, you need to prepare the data, then fire up the Explorer and load in the data. Next you select a decision tree construction method, build a tree, and interpret the output. It is easy to do it again with a different tree construction algorithm or a different evaluation method. In the Explorer you can flip back and forth between the results you have obtained, evaluate the models that have been built on different datasets, and visualize graphically both the models and the datasets themselves—including any classification errors the models make.

(a) Spreadsheet.

(c) ARFF.

Figure 2.2: Weather data.

(b) CSV.

2.1. GETTING STARTED 17

**2.1.1 Preparing the data**

The data is often presented in a spreadsheet or database. However, WEKA’s native data storage method is ARFF format. You can easily convert from a spreadsheet to ARFF. The bulk of an ARFF file consists of a list of the instances, and the attribute values for each instance are separated by commas. Most spreadsheet and database programs allow you to export data into a file in comma-separated value (CSV) format as a list of records with commas between items. Having done this, you need only load the file into a text editor or word processor; add the dataset’s name using the @relation tag, the attribute information using @attribute, and a @data line; and save the file as raw text. For example, Figure 2.19 shows an Excel spreadsheet containing the weather data, the data in CSV form loaded into Microsoft Word, and the result of converting it manually into an ARFF file. However, you do not actually have to go through these steps to create the ARFF file yourself, because the Explorer can read CSV spreadsheet files directly, as described later.

**2.1.2 Loading the data into the Explorer**

(b) Reading in the weather data.

(a) Choosing the Explorer inter- face.

Figure 2.3: Weather data in the Explorer.

Let us load this data into the Explorer and start analyzing it. Fire up WEKA to get the *GUI Chooser panel* in Figure 2.3a. Select *Explorer* from the five choices on the right-hand side. (The others were mentioned earlier: *Simple CLI* is the old-fashioned command-line interface.)

What you see next is the main Explorer screen, shown in Figure 2.3b. Actually, the figure shows what it will look like *after* you have loaded in the weather data. The six tabs along the top are the basic operations that the Explorer supports: right now we are on *Preprocess*. Click the Open file button to bring up a standard dialog through which you can select a file. Choose the *weather.arff* file. If you have it in CSV format, change from *ARFF data files* to *CSV data files*. When you specify a .csv file it is automatically converted into ARFF format.

Having loaded the file, the screen will be as shown in Figure 2.3b. This tells you about the dataset: it has 14 instances and five attributes (center left); the attributes are called *outlook*,

18 CHAPTER 2. THE EXPLORER

*temperature*, *humidity*, *windy*, and *play* (lower left). The first attribute, *outlook*, is selected by default (you can choose others by clicking them) and has no missing values, three distinct values, and no unique values; the actual values are *sunny*, *overcast*, and *rainy* and they occur five, four, and five times, respectively (center right). A histogram at the lower right shows how often each of the two values of the class, *play*, occurs for each value of the *outlook* attribute. The attribute *outlook* is used because it appears in the box above the histogram, but you can draw a histogram of any other attribute instead. Here *play* is selected as the class attribute; it is used to color the histogram, and any filters that require a class value use it too.

The *outlook* attribute in Figure 2.3b is nominal. If you select a numeric attribute, you see its minimum and maximum values, mean, and standard deviation. In this case the histogram will show the distribution of the class as a function of this attribute.

You can delete an attribute by clicking its checkbox and using the *Remove* button. *All* selects all the attributes, *None* selects none, Invert *inverts* the current selection, and *Pattern* selects those attributes whose names match a user-supplied regular expression. You can undo a change by clicking the *Undo* button. The *Edit* button brings up an editor that allows you to inspect the data, search for particular values and edit them, and delete instances and attributes. Right-clicking on values and column headers brings up corresponding context menus.

**2.1.3 Building a decision tree**

(b) The *Classify* tab.

Figure 2.4: J4.8 in the Explorer.

(a) Finding J4.8 in the classifiers list.

To see what the C4.5 decision tree learner does with this dataset, use the J4.8 algorithm, which is WEKA’s implementation of this algorithm. (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 was released.) Click the *Classify* tab to get a screen that looks like Figure 2.4b. Actually, the figure shows what it will look like *after* you have analyzed the weather data.First select the classifier by clicking the *Choose* button at the top left, opening up the trees section of the hierarchical menu in Figure 2.4a, and finding *J48*. The menu structure represents

2.1. GETTING STARTED 19

the organization of the WEKA code into modules and the items you need to select are always at the lowest level. Once selected, *J48* appears in the line beside the *Choose* button as shown in Figure 2.4b, along with its default parameter values. If you click that line, the J4.8 classifier’s object editor opens up and you can see what the parameters mean and alter their values if you wish. The Explorer generally chooses sensible defaults.

Having chosen the classifier, invoke it by clicking the *Start* button. WEKA works for a brief period—when it is working, the little bird at the lower right of Figure 2.4b jumps up and dances—and then produces the output shown in the main panel of Figure 2.4b.

**2.1.4 Examining the output**

Figure 2.5 shows the full output (Figure 2.4b only gives the lower half). At the beginning is a summary of the dataset, and the fact that 10-fold cross-validation was used to evaluate it. That is the default, and if you look closely at Figure 2.4b you will see that the *Cross-validation* box at the left is checked. Then comes a pruned decision tree in textual form. The model that is shown here is always one generated from the full dataset available from the *Preprocess panel*. 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 decimal number because of the way the algorithm uses fractional instances to handle missing values. If there were incorrectly classified instances (there aren’t in this example) their number would appear too: thus *2.0/1.0* means that two instances reached that leaf, of which one is classified incorrectly. Beneath the tree structure the number of leaves is printed; then the total number of nodes (*Size of the tree*). There is a way to view decision trees more graphically, which we will encounter later.

The next part of the output gives estimates of the tree’s predictive performance. In this case they are obtained using stratified cross-validation with 10 folds, the default in Figure 2.4b. 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 optimistic compared with what might be obtained from an independent test set from the same source. From the confusion matrix at the end observe that 2 instances of class *yes* have been assigned to class *no* and 3 of class *no* are assigned to class *yes*.

As well as the classification error, the evaluation module also outputs the Kappa statistic, the mean absolute error, and the root mean-squared error of the class probability estimates assigned by the tree. The root mean-squared error is the square root of the average squared loss. The mean absolute error is calculated in a similar way using the absolute instead of the squared difference. It also outputs relative errors, which are based on the prior probabilities (i.e., those obtained by the ZeroR learning scheme described later). Finally, for each class it also outputs various statistics. Also reported is the per-class average of each statistic, weighted by the number of instances from each class. All of these evaluation measures are discussed in Chapter 5 of the book.

**2.1.5 Doing it again**

You can easily run J4.8 again with a different evaluation method. Select *Use training set* (near the top left in Figure 2.4b) and click *Start* again. The classifier output is quickly replaced to show how well the derived model performs on the training set, instead of showing the cross-validation results. This evaluation is highly optimistic. It may still be useful, because it generally represents an upper bound to the model’s performance on fresh data. In this case, all 14 training instances

20 CHAPTER 2. THE EXPLORER

=== Run information ===

Scheme: weka.classifiers.trees.J48 -C 0.25 -M 2 Relation: weather Instances: 14 Attributes: 5outlook

temperature humidity windy play Test mode: 10-fold cross-validation

=== Classifier model (full training set) ===

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

Time taken to build model: 0.27 seconds

=== Stratified cross-validation === === Summary ===

Correctly Classified Instances 9 64.2857 % Incorrectly Classified Instances 5 35.7143 % Kappa statistic 0.186 Mean absolute error 0.2857 Root mean squared error 0.4818 Relative absolute error 60 % Root relative squared error 97.6586 % Total Number of Instances 14

=== Detailed Accuracy By Class ===

TP Rate FP Rate Precision Recall F-Measure ROC Area Class

0.778 0.6 0.7 0.778 0.737 0.789 yes 0.4 0.222 0.5 0.4 0.444 0.789 no Weighted Avg. 0.643 0.465 0.629 0.643 0.632 0.789

=== Confusion Matrix ===

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

Figure 2.5: Output from the J4.8 decision tree learner.

2.1. GETTING STARTED 21

are classified correctly. In some cases a classifier may decide to leave some instances unclassified, in which case these will be listed as *Unclassified Instances*. This does not happen for most learning schemes in WEKA.

The panel in Figure 2.4b has further test options: *Supplied test set*, in which you specify a separate file containing the test set, and *Percentage split*, with which you can hold out a certain percentage of the data for testing. You can output the predictions for each instance by clicking the *More options* button and checking the appropriate entry. There are other useful options, such as suppressing some output and including other statistics such as entropy evaluation measures and cost-sensitive evaluation. For the latter you must enter a cost matrix: type the number of classes into the *Classes* box (and terminate it with the *Enter* or *Return* key) to get a default cost matrix, then edit the values as required.

**2.1.6 Working with models**

The small pane at the lower left of Figure 2.4b, which contains one highlighted line, is a history list of the results. The Explorer adds a new line whenever you run a classifier. Because you have now run the classifier twice, the list will contain two items. To return to a previous result set, click the corresponding line and the output for that run will appear in the Classifier Output pane. This makes it easy to explore different classifiers or evaluation schemes and revisit the results to compare them.

The result history list is the entry point to some powerful features of the Explorer. When you *right*-click an entry a menu appears that allows you to view the results in a separate window, or save the result buffer. More importantly, you can save the model that WEKA has generated in the form of a Java object file. You can reload a model that was saved previously, which generates a new entry in the result list. If you now supply a test set, you can reevaluate the old model on that new set.

Several items on the right-click menu allow you to visualize the results in various ways. At the top of the Explorer interface is a separate *Visualize* tab, but that is different: it shows the dataset, not the results for a particular model. By right-clicking an entry in the history list you can see the classifier errors. If the model is a tree or a Bayesian network you can see its structure. You can also view the margin curve and various cost and threshold curves, including the cost/benefit analysis tool. For all of these you must choose a class value from a submenu. The *Visualize threshold curve* menu item allows you to see the effect of varying the probability threshold above which an instance is assigned to that class. You can select from a wide variety of curves that include the ROC and recall-precision curves. To see these, choose the X- and Y-axes appropriately from the menus given. For example, set X to *False Positive Rate* and Y to *True Positive Rate* for an ROC curve or X to *Recall* and Y to *Precision* for a recall-precision curve.

Figure 2.6 shows two ways of looking at the result of using J4.8 to classify the iris dataset—we use this rather than the weather data because it produces more interesting pictures. Figure 2.6a shows the tree. Right-click a blank space in this window to bring up a menu enabling you to automatically scale the view or force the tree into the window. Drag the mouse to pan around the space. It’s also possible to visualize the instance data at any node, if it has been saved by the learning algorithm.

Figure 2.6b shows the classifier errors on a two-dimensional plot. You can choose which attributes to use for X and Y using the selection boxes at the top. Alternatively, click one of the speckled horizontal strips to the right of the plot: left-click for X and right-click for Y. Each strip shows the spread of instances along that attribute. X and Y appear beside the ones you have chosen for the axes.

The data points are colored according to their class: blue, red and green for *Iris setosa*, *Iris*

22 CHAPTER 2. THE EXPLORER

*versicolor*, and *Iris virginica*, respectively (there is a key at the bottom of the screen). Correctly classified instances are shown as crosses; incorrectly classified ones appear as boxes (of which there are three in Figure 2.6b). You can click on an instance to bring up relevant details: its instance number, the values of the attributes, its class, and the predicted class.

**2.1.7 When things go wrong**

Beneath the result history list, at the bottom of Figure 2.3b, is a status line that says, simply, *OK*. Occasionally this changes to *See error log*, an indication that something has gone wrong. For example, there may be constraints among the various different selections you can make in a panel. Most of the time the interface grays out inappropriate selections and refuses to let you choose them. But occasionally the interactions are more complex, and you can end up selecting an incompatible set of options. In this case, the status line changes when WEKA discovers the incompatibility—typically when you press *Start*. To see the error, click the *Log* button to the left of the bird in the lower right-hand corner of the interface. WEKA also writes a detailed log to a file, called *weka.log*, under the *wekafiles* directory in the user’s home directory. This often contains more information about the causes of problems than the Explorer’s Log window because it captures debugging output directed to the standard out and error channels.

**2.2 Exploring the Explorer**

We have briefly investigated two of the six tabs at the top of the Explorer window in Figure 2.3b and Figure 2.4b. In summary, here’s what all of the tabs do:

1. *Preprocess*: Choose the dataset and modify it in various ways.

2. *Classify*: Train learning schemes that perform classification or regression and evaluate

them.

3. *Cluster*: Learn clusters from the dataset.

4. *Associate*: Learn association rules for the data and evaluate them.

5. *Select attributes*: Select the most relevant aspects of the dataset.

6. *Visualize*: View different two-dimensional plots of the data and interact with them.

Each tab gives access to a whole range of facilities. In our tour so far, we have barely scratched the surface of the *Preprocess* and *Classify* panels.

At the bottom of every panel is a *Status* box and a *Log* button. The status box displays messages that keep you informed about what’s going on. For example, if the Explorer is busy loading a file, the status box will say so. Right-clicking anywhere inside this box brings up a little menu with two options: display the amount of memory available to WEKA, and run the Java garbage collector. Note that the garbage collector runs constantly as a background task anyway.

Clicking the *Log* button opens a textual log of the actions that WEKA has performed in this session, with timestamps.

As noted earlier, the little bird at the lower right of the window jumps up and dances when WEKA is active. The number beside the × shows how many concurrent processes are running. If the bird is standing but stops moving, it’s sick! Something has gone wrong, and you may have to restart the Explorer.

2.2. EXPLORING THE EXPLORER 23

**2.2.1 Loading and filtering files**

Along the top of the *Preprocess panel* in Figure 2.3b are buttons for opening files, URLs, and databases. Initially, only files whose names end in *.arff* appear in the file browser; to see others, change the *Format* item in the file selection box.

**2.2.2 Converting files to ARFF**

WEKA has converters for the following file formats:

• spreadsheet files with extension *.csv*,

• C4.5’s native file format with extensions *.names* and *.data*,

• serialized instances with extension *.bsi*,

• LIBSVM format files with extension *.libsvm*,

• SVM-Light format files with extension *.dat*,

• XML-based ARFF format files with extension *.xrff*,

• JSON-based ARFF format files with extension *.json*,

• ASCII Matlab files with extension *.m*.

The appropriate converter is used based on the file extension. If WEKA cannot load the data, it tries to interpret it as ARFF. If that fails, it pops up the box shown in Figure 2.7a.

This is a generic object editor, used throughout WEKA for selecting and configuring object. For example, when you set parameters for a classifier, you use the same kind of box. The *CSVLoader* for *.csv* files is selected by default, and the *More* button gives you more information about it, shown in Figure 2.7b. It is always worth looking at the documentation! In this case, it explains that the spreadsheet’s first row determines the attribute names and gives a brief description of the CSVLoader’s options. Click *OK* to use this converter. For a different one, click *Choose* to select from the list in Figure 2.7c.

The *ArffLoader* is the first option, and we reached this point only because it failed. The second option is for the C4.5 format, in which there are two files for a dataset, one giving field names and the other giving the actual data. The third option, *CSVLoader*, is the default, and we clicked *Choose* because we want a different one. The fourth option is for reading from a database rather than a file; however, the *SQLViewer* tool, shown in Figure 2.8 and accessible by pressing the *Open DB* button on the *Preprocess* panel, is a more user-friendly route for accessing a database. The “serialized instances” option is for reloading datasets that have been saved as Java serialized object. Any Java object can be saved in this format and reloaded. As a native Java format, it is quicker to load that an ARFF file, which must be parsed and checked. When repeatedly reloading a large dataset it may be worth saving it in this form.

The tenth menu item is for importing a directory containing plain text files for the purposes of text mining. The imported directory is expected to have a specific structure—namely a set of subdirectories, each containing one or more text files with the extension *.txt*. Each text file becomes one instance in the dataset, with a string attribute holding the contents of the file and a nominal class attribute holding the name of the subdirectory that it came from. This dataset can then be further processed into word frequencies using the *StringToWordVector* filter (covered in the next section). The last option is for loading data files in XRFF, the XML Attribute Relation

24 CHAPTER 2. THE EXPLORER

File format. As the name suggests, this gives ARFF header and instance information in the XML markup language.

Further features of the generic object editor in Figure 2.7a are *Save*, which saves a configured object, and *Open*, which opens a previously saved one. These are not useful for this particular kind of object. But other generic object editor panels have many editable properties, and having gone to some trouble to set them up you may want to save the configured object to reuse later. Files on your computer are not the only source of datasets for WEKA. You can open a URL, and WEKA will use the hypertext transfer protocol (HTTP) to download an ARFF file from the Web. Or you can open a database (Open DB)—any database that has a Java database connectivity (JDBC) driver—and retrieve instances using the SQL *Select* statement. This returns a relation that WEKA reads in as an ARFF file. To make this work with your database, you may need to modify the file *weka/experiment/DatabaseUtils.props* in the WEKA distribution by adding your database driver to it. (To access this file, expand the weka.jar file in the WEKA distribution.) Figure 2.8 shows the SQLViewer tool that appears when *Open DB* is clicked. In this example, the iris dataset has been extracted from a single database table.

Data can be saved in all these formats (with the exception of the directory containing text files) using the Save button in the *Preprocess* panel (Figure 2.3b). It is also possible to generate artificial data using the *Generate* button. Artificial data suitable for classification can be gen- erated from decision lists, radial-basis function networks and Bayesian networks, as well as the classic LED24 domain. Artificial regression data can be generated according to mathematical expressions. There are also several generators for producing artificial data for clustering pur- poses. Apart from loading and saving datasets, the *Preprocess* panel also allows you to filter them. Filters are an important component of WEKA.

**2.2.3 Using filters**

Clicking *Choose* (near the top left) in Figure 2.3b gives a list of filters like that in Figure 2.9a. Actually, you get a collapsed version: click on an arrow to open up its contents. We will describe how to use a simple filter to delete specified attributes from a dataset, in other words, to perform manual attribute selection. The same effect can be achieved more easily by selecting the relevant attributes using the tick boxes and pressing the *Remove* button. Nevertheless we describe the equivalent filtering operation explicitly, as an example.

*Remove* is an unsupervised attribute filter, and to see it you must scroll further down the list. When selected, it appears in the line beside the *Choose* button, along with its parameter values—in this case the line reads simply “Remove”. Click that line to bring up a generic object editor with which you can examine and alter the filter’s properties. (You did the same thing earlier by clicking the *J48* line in Figure 2.4b to open the *J4.8* classifier’s object editor.) The object editor for the Remove filter shown in Figure 2.9b.

To learn about it, click *More* to show the information in Figure 2.9c. This explains that the filter removes a range of attributes from the dataset. It has an option, *attributeIndices*, that specifies the range to act on and another called *invertSelection* that determines whether the filter selects attributes or deletes them. There are boxes for both of these in the object editor shown in Figure 2.9b, and in fact we have already set them to *1,2* (to affect attributes 1 and 2, namely *outlook* and *temperature*) and *False* (to remove rather than retain them). Click *OK* to set these properties and close the box. Notice that the line beside the *Choose* button now reads *Remove –R 1,2*. In the command-line version of the *Remove* filter, the option *–R* is used to specify which attributes to remove. After configuring an object it’s often worth glancing at the resulting command-line formulation that the Explorer sets up.

Figure 2.9 demonstrates a further feature of the generic object editor, namely “capabilities.”

2.2. EXPLORING THE EXPLORER 25

Algorithms in WEKA may provide information about what data characteristics they can handle, and, if they do, a *Capabilities* button appears underneath *More* in the generic object editor (Figure 2.9b). Clicking it brings up Figure 2.9d, which gives information about what the method can do. Here it states that *Remove* can handle many attribute characteristics, such as different types (nominal, numeric, relational, etc.) and missing values. It shows the minimum number of instances that are required for *Remove* to operate on.

Figure 2.9e shows a list of selected constraints on capabilities, which is obtained by clicking the *Filter* button at the bottom of Figure 2.9a. If the current dataset exhibits some characteristic that is ticked in Figure 2.9e but missing from the capabilities for the *Remove* filter (Figure 2.9d), the *Apply* button to the right of *Choose* in Figure 2.3b will be grayed out, as will the entry in the list in Figure 2.9a. Although you cannot apply it, you can nevertheless select a grayed-out entry to inspect its options, documentation and capabilities using the generic object editor. You can release individual constraints by deselecting them in Figure 2.9e, or click the *Remove filter* button to clear all the constraints.

Apply the filter by clicking *Apply* (at the right-hand side of Figure 2.3b). Immediately the screen in Figure 2.10 appears—just like the one in Figure 2.3b but with only three attributes, *humidity*, *windy*, and *play*. At this point the fifth button in the row near the top becomes active. *Undo* reverses the filtering operation and restores the original dataset, which is useful when you experiment with different filters.

The first attribute, *humidity*, is selected and a summary of its values appears on the right. As a numeric attribute, the minimum and maximum values, mean, and standard deviation are shown. Below is a histogram that shows the distribution of the *play* attribute. Unfortunately, this display is impoverished because the attribute has so few different values that they fall into two equal-sized bins. More realistic datasets yield more informative histograms.

**2.2.4 Training and testing learning schemes**

The *Classify* panel lets you train and test learning schemes that perform classification or regres- sion. Section 2.1 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 interpreta- tion 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, in Figure 2.11a the CPU performance dataset has been loaded into WEKA. You can see the histogram of values of the first attribute, *MYCT*, at the lower right. In Fig- ure 2.11b the model tree inducer M5’ has been chosen as the classifier by going to the *Classify* panel, clicking the *Choose* button at the top left, opening up the trees section of the hierarchi- cal menu shown in Figure 2.4a, finding *M5P*, and clicking *Start*. The hierarchy helps to locate particular classifiers by grouping items with common functionality

Figure 2.12 shows the output. The pruned model tree contains splits on three of the six attributes in the data. The root splits on the *CHMIN* attribute, yielding a linear model at the leaf on the left-hand branch and the remaining structure in the right-hand branch. There are five leaves in all, each with a corresponding linear model. The first number in parentheses at each leaf is the number of instances that reach it; the second is the root mean squared error of the predictions from the leaf’s linear model for those instances, expressed as a percentage of the standard deviation of the class attribute computed over all the training data. The description of the tree is followed by several figures that measure its performance. These are derived from the test option chosen in Figure 2.11b, 10-fold cross-validation (not stratified, because stratification does not apply when the class is numeric).

Ordinary linear regression, another scheme for numeric prediction, is found under *Linear-*

26 CHAPTER 2. THE EXPLORER

*Regression* in the *functions* section of the menu in Figure 2.4a. It builds a single linear regression model rather than the five in Figure 2.12; not surprisingly, its performance is slightly worse.

To get a feel for their relative performance, let us visualize the errors these schemes make, as we did for the iris dataset in Figure 2.6b. Right-click the entry in the history list and select *Visualize classifier errors* to bring up the two-dimensional plot of the data in Figure 2.16. The points are color coded by class—but in this case the color varies continuously because the class is numeric. In Figure 2.16 the *MMAX* attribute has been selected for the X-axis and *CHMAX* has been chosen for the Y-axis because this gives a reasonable spread of points. Each data point is marked by a cross whose size indicates the absolute value of the error for that instance. The smaller crosses in Figure 2.13a (for M5’), when compared with those in Figure 2.13b (for linear regression), show that M5’ is superior.

**2.2.5 Using a metalearner**

Metalearners take simple classifiers and turn them into more powerful learners. For example, to boost decision stumps in the Explorer, go to the *Classify* panel and choose the classifier *AdaboostM1* from the meta section of the hierarchical menu. When you configure this classifier by clicking it, the object editor shown in Figure 2.14 appears. This has its own classifier field, which we set to *DecisionStump* (as shown). This method could itself be configured by clicking (except that *DecisionStump* happens to have no editable properties). Click *OK* to return to the main *Classify* panel and *Start* to try out boosting decision stumps up to 10 times. It turns out that this mislabels only 7 of the 150 instances in the iris data—good performance considering the rudimentary nature of decision stumps and the rather small number of boosting iterations.

**2.2.6 Clustering and association rules**

Use the *Cluster* and *Associate* panels to invoke clustering algorithms and methods for finding association rules. When clustering, WEKA shows the number of clusters and how many instances each cluster contains. For some algorithms the number of clusters can be specified by setting a parameter in the object editor. For probabilistic clustering methods, WEKA measures the log- likelihood of the clusters on the training data: the larger this quantity, the better the model fits the data. Increasing the number of clusters normally increases the likelihood, but may overfit.

The controls on the *Cluster* panel are similar to those for *Classify*. You can specify some of the same evaluation methods—use training set, supplied test set, and percentage split (the last two are used with the log-likelihood). A further method, classes to clusters evaluation, compares how well the chosen clusters match a preassigned class in the data. You select an attribute (which must be nominal) that represents the “true” class. Having clustered the data, WEKA determines the majority class in each cluster and prints a confusion matrix showing how many errors there would be if the clusters were used instead of the true class. If your dataset has a class attribute, you can ignore it during clustering by selecting it from a pull-down list of attributes, and see how well the clusters correspond to actual class values. Finally, you can choose whether or not to store the clusters for visualization. The only reason not to do so is to conserve space. As with classifiers, you visualize the results by right-clicking on the result list, which allows you to view two-dimensional scatter plots like the one in Figure 2.6b. If you have chosen classes to clusters evaluation, the class assignment errors are shown. For the *Cobweb* clustering scheme, you can also visualize the tree.

The *Associate* panel is simpler than *Classify* or *Cluster*. WEKA contains several algorithms for determining association rules and no methods for evaluating such rules. Figure 2.15 shows the output from the Apriori program for association rules on the nominal version of the weather

2.3. FILTERING ALGORITHMS 27

data. Despite the simplicity of the data, several rules are found. The number before the arrow is the number of instances for which the antecedent is true; that after the arrow is the number of instances for which the consequent is true also; and the confidence (in parentheses) is the ratio between the two. Ten rules are found by default: you can ask for more by using the object editor to change *numRules*.

**2.2.7 Attribute selection**

The *Select attributes* panel gives access to several methods for attribute selection. This involves an attribute evaluator and a search method. Both are chosen in the usual way and configured with the object editor. You must also decide which attribute to use as the class. Attribute selection can be performed using the full training set or using cross-validation. In the latter case it is done separately for each fold, and the output shows how many times—that is, in how many of the folds—each attribute was selected. The results are stored in the history list. When you right-click an entry here you can visualize the dataset in terms of the selected attributes (choose *Visualize reduced data*).

**2.2.8 Visualization**

The *Visualize* panel helps you visualize a dataset—not the result of a classification or clustering model, but the dataset itself. It displays a matrix of two-dimensional scatter plots of every pair of attributes. Figure 2.16a shows the iris dataset. You can select an attribute—normally the class—for coloring the data points using the controls at the bottom. If it is nominal, the coloring is discrete; if it is numeric, the color spectrum ranges continuously from blue (low values) to orange (high values). Data points with no class value are shown in black. You can change the size of each plot, the size of the points, and the amount of jitter, which is a random displacement applied to X and Y values to separate points that lie on top of one another. Without jitter, a thousand instances at the same data point would look just the same as one instance. You can reduce the size of the matrix of plots by selecting certain attributes, and you can subsample the data for efficiency. Changes in the controls do not take effect until the *Update* button is clicked. Click one of the plots in the matrix to enlarge it. For example, clicking on the top left plot brings up the panel in Figure 2.16b. You can zoom in on any area of this panel by choosing *Rectangle* from the menu near the top right and dragging out a rectangle on the viewing area like that shown. The *Submit* button near the top left rescales the rectangle into the viewing area.

**2.3 Filtering algorithms**

Now we take a detailed look at the filtering algorithms implemented within WEKA. These are accessible from the Explorer, and also from the Knowledge Flow and Experimenter interfaces described later. All filters transform the input dataset in some way. When a filter is selected using the *Choose* button, its name appears in the line beside that button. Click that line to get a generic object editor to specify its properties. What appears in the line is the command-line version of the filter, and the parameters are specified with minus signs. This is a good way of learning how to use the WEKA commands directly.

There are two kinds of filter: unsupervised and supervised. This seemingly innocuous dis- tinction masks a rather fundamental issue. Filters are often applied to a training dataset and then also applied to the test file. If the filter is supervised—for example, if it uses class values to derive good intervals for discretization—applying it to the test data will bias the results. It is the discretization intervals derived from the *training* data that must be applied to the test data.

28 CHAPTER 2. THE EXPLORER

When using supervised filters you must to be careful to ensure that the results are evaluated fairly, an issue that does not arise with unsupervised filters.

We treat WEKA’s unsupervised and supervised filtering methods separately. Within each type there is a further distinction between *attribute filters*, which work on the attributes in the datasets, and *instance filters*, which work on the instances. To learn more about a particular filter, select it in the WEKA Explorer and look at its associated object editor, which defines what the filter does and the parameters it takes.

**2.3.1 Unsupervised attribute filters**

We will first discuss WEKA’s unsupervised attribute filters, which do not require a class attribute to be set.

**2.3.1.1 Adding and removing attributes**

*Add* inserts an attribute at a given position, whose value is declared to be missing for all instances. Use the generic object editor to specify the attribute’s name, where it will appear in the list of attributes, and its possible values (for nominal attributes); for date attributes you can also specify the date format. *Copy* copies existing attributes so that you can preserve them when experimenting with filters that overwrite attribute values. Several attributes can be copied together using an expression such as *1âĂŞ3* for the first three attributes, or *first-3,5,9-last* for attributes 1, 2, 3, 5, 9, 10, 11, 12 .... The selection can be inverted, affecting all attributes except those specified. These features are shared by many filters. *AddUserFields* can be used to add multiple new attributes to a dataset in one go. The user can specify the name, type and value to use for each attribute.

*AddID* inserts a numeric identifier attribute at the user-specified index in the list of attributes. An identifier attribute is useful for keeping track of individual instances after a dataset has been processed in various ways, such as being transformed by other filters, or having the order of the instances randomized.

*Remove* has already been described. Similar filters are *RemoveType*, which deletes all at- tributes of a given type (nominal, numeric, string, date, or relational), and *RemoveUseless*, which deletes constant attributes and nominal attributes whose values are different for almost all instances. You can decide how much variation is tolerated before an attribute is deleted by specifying the number of distinct values as a percentage of the total number of values. Note that some unsupervised attribute filters behave differently if the menu in the *Preprocess* panel has been used to set a class attribute. (By default, the last attribute is the class attribute.) For example, *RemoveType* and *RemoveUseless* both skip the class attribute.

*InterquartileRange* adds new attributes that indicate whether the values of instances can be considered outliers, or extreme values. The definition of outlier and extreme value is based on the difference between the 25th and 75th quartile of an attribute’s values. Values are flagged as extreme if they exceed the 75th quartile (or fall below the 25th quartile) by the product of the user-specified extreme value factor and the interquartile range. Values that are not extreme values but exceed the 75th quartile (or fall below the 25th quartile) by the product of the outlier factor and the interquartile range are flagged as outliers. The filter can be configured to flag an instance as an outlier or extreme if any of its attribute values are deemed outliers or extreme, or to generate an outlier/extreme indicator pair for each attribute. It is also possible to flag all extreme values as outliers, and to output attributes that indicate by how many interquartile ranges an attribute’s value deviates from the median.

*AddCluster* applies a clustering algorithm to the data before filtering it. You use the object editor to choose the clustering algorithm. Clusterers are configured just as filters are. The

2.3. FILTERING ALGORITHMS 29

*AddCluster* object editor contains its own *Choose* button for the clusterer, and you configure the clusterer by clicking its line and getting another object editor panel, which must be filled in before returning to the *AddCluster* object editor. This is probably easier to understand when you do it in practice than when you read about it here! At any rate, once you have chosen a clusterer, *AddCluster* uses it to assign a cluster number to each instance, as a new attribute. The object editor also allows you to ignore certain attributes when clustering, specified as described previously for *Copy*. *ClusterMembership* uses a clusterer, again specified in the filter’s object editor, to generate membership values. A new version of each instance is created whose attributes are these values. The class attribute, if set, is ignored during clustering.

*AddExpression* creates a new attribute by applying a mathematical function to numeric at- tributes. The expression can contain attribute references and constants; arithmetic operators +, –,\*, /, and ˆ; the functions *log* and *exp*, *abs* and *sqrt*, *floor*, *ceil* and *rint*1, *sin*, *cos* and *tan*; and parentheses. Attributes are specified by the prefix *a*, for example, *a7* is the seventh attribute. An example expression is

a1ˆ2\*a5/log(a7\*4.0)

There is a debug option that replaces the new attribute’s value with a postfix parse of the supplied expression.

*MathExpression* is similar to *AddExpression* but can be applied to multiple attributes. Rather than creating a new attribute it replaces the original values with the result of the expression in *situ*; because of this, the expression cannot reference the value of other attributes. All the operators that apply to *AddExpression* can be used, as well as the minimum, maximum, mean, sum, sum-squared and standard deviation of the attribute being processed. Furthermore, simple if-then-else expressions involving the operators and functions can be applied as well.

Whereas *AddExpression* and *MathExpression* apply mathematical functions specified in tex- tual form, *NumericTransform* performs an arbitrary transformation by applying a given Java function to selected numeric attributes. The function can be anything that takes a *double* as its argument and returns another *double*, for example, *sqrt()* in *java.lang.Math*. One parameter is the name of the Java class that implements the function (which must be a fully qualified name); another is the name of the transformation method itself.

*Normalize* scales all numeric values in the dataset to lie between 0 and 1. The normalized values can be further scaled and translated with user-supplied constants. *Center* and *Standardize* transform them to have zero mean; *Standardize* gives them unit variance too. All three skip the class attribute, if set. *RandomSubset* randomly selects a subset of the attributes to include in the output; the user specifies how many (as a percentage of the number of attributes). The class attribute, if set, is always included.

*CartesianProduct* produces a new attribute with values resulting from performing the Carte- sian product between two or more nominal attributes. The name of the new attribute is a concatenation of the names of the original attributes.

*PartitionedMultiFilter* is a special filter that applies a set of filters to a corresponding set of attribute ranges in the input dataset. The user supplies and configures each filter, and defines the range of attributes for them to work with. There is an option to delete attributes that are not covered by any of the ranges. Only filters that operate on attributes are allowed. The output of the individual filters is assembled into a new dataset.

*Reorder* alters the order of the attributes in the data; the new order is specified by supplying a list of attribute indices. By omitting or duplicating indices it is possible to delete attributes or make several copies of them.

1The *rint* function rounds to the closest integer.

30 CHAPTER 2. THE EXPLORER

**2.3.1.2 Changing values**

*SwapValues* swaps the positions of two values of a nominal attribute. The order of values is entirely cosmetic—it does not affect learning at all—but if the class is selected, changing the order affects the layout of the confusion matrix. *MergeTwoValues* merges values of a nominal attribute into a single category. The new value’s name 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 indices. For example, if you merge the first two values of the *outlook* attribute in the weather data—in which there are five *sunny*, four *overcast*, and five *rainy* instances—the new *outlook* attribute will have values *sunny overcast* and *rainy*; there will be nine *sunny overcast* instances and the original five *rainy* ones. *MergeManyValues* is similar to *MergeTwoValues* but allows any number of values of a nominal attribute to be replaced by a single category. The user can specify the name of the new category to create. *MergeInfrequentValues* also merges several nominal values into one category, but in this case the process is controlled by a user-supplied minimum frequency. Values that occur fewer times than this minimum are replaced with the new category. Like *SwapValues*, the new value’s name is a concatenation of the the original ones. The user can opt to have a short identifier, based on a hash code, used as the new value instead—this is useful if the concatenated names become very long.One way of dealing with missing values is to replace them globally before applying a learning scheme. *ReplaceMissingValues* replaces each missing value by the mean for numeric attributes and the mode for nominal ones. If a class is set, missing values of that attribute are not replaced by default, but this can be changed. *ReplaceMissingWithUserConstant* is another filter that can replace missing values. In this case, it allows the user to specify a constant value to use. This constant can be specified separately for numeric, nominal and date attributes.

*ReplaceWithMissingValue* is a filter that does the opposite of the ones we have just discussed— it replaces non-missing values, for a user-specified range of attributes, with missing values at random. The probability that a given value will be replaced with a missing value can be set as an option.

*NumericCleaner* replaces the values of numeric attributes that are too small, too large or too close to a particular value with default values. A different default can be specified for each case, along with thresholds for what is considered to be too large or small and a tolerance value for defining “too close” can be specified as well.

*AddValues* adds any values from a user-supplied list that are not already present in a nominal attribute. The labels can optionally be sorted. *ClassAssigner* can be used to set or unset a dataset’s class attribute. The user supplies the index of the new class attribute; a value of zero unsets the existing class attribute.

*SortLabels* can be used to sort the values of selected nominal attributes.

**2.3.1.3 Conversions**

Many filters convert attributes from one form to another. *Discretize* uses equal-width or equal- frequency binning to discretize a range of numeric attributes, specified in the usual way. For the former method the number of bins can be specified or chosen automatically by maximizing the likelihood using leave-one-out cross-validation. It is also possible to create several binary attributes instead of one multi-valued one. For equal-frequency discretization, the desired number of instances per interval can be changed. *PKIDiscretize* discretizes numeric attributes using equal-frequency binning; the number of bins is the square root of the number of values (excluding missing values). Both these filters skip the class attribute by default.

2.3. FILTERING ALGORITHMS 31

*MakeIndicator* converts a nominal attribute into a binary indicator attribute and can be used to transform a multiclass dataset into several two-class ones. It substitutes a binary attribute for the chosen nominal one, whose values for each instance are 1 if a particular original value was present and 0 otherwise. The new attribute is declared to be numeric by default, but it can be made nominal if desired.

For some learning schemes, such as support vector machines, multivalued nominal attributes must be converted to binary ones. The *NominalToBinary* filter transforms all specified multival- ued nominal attributes in a dataset into binary ones, replacing each attribute with *k* values by *k* binary attributes using a simple one-per-value (“one-hot”) encoding. The new attributes will be numeric by default. Attributes that are already binary are left untouched. *NumericToBinary* converts all numeric attributes into nominal binary ones (except the class, if set). If the value of the numeric attribute is exactly 0, the new attribute will be 0, and if it is missing, the new at- tribute will be missing; otherwise, the value of the new attribute will be 1. These filters also skip the class attribute. *NumericToNominal* converts numeric attributes to nominal ones by simply adding every distinct numeric value to the list of nominal values. This can be a useful filter to apply after importing a .csv file—WEKA’s CSV import facility creates a numeric attribute for any data column whose values can all be parsed as numbers, unless configured to do otherwise, but it might make sense to interpret the values of an integer attribute as discrete instead.

*FirstOrder* takes a range of *N* numeric attributes and replaces them with *N* – 1 numeric attributes whose values are the differences between consecutive attribute values from the original instances. For example, if the original attribute values were 3, 2, and 1, the new ones will be –1 and –1.

*KernelFilter* converts the data to a kernel matrix: it outputs a new dataset, containing the same number of instances as before, in which each value is the result of evaluating a kernel function on a pair of the original instances. By default, all attributes are transformed to center them around zero before the kernels are computed, though they are not rescaled to unit variance. However, different filters can be specified.

*PrincipalComponents* performs a principal components transformation on the dataset. First, any multi-valued nominal attributes are converted to binary, and missing values are replaced by means. The data is normalized (by default). The number of components is normally determined based on the user-specified proportion of variance to be covered, but it is also possible to specify the number of components explicitly.

*Transpose*, as the name suggests, transposes the data—instances become attributes and at- tributes become instances. If the first attribute in the original data is a nominal or string identifier, this will be used to create attribute names in the transposed data. Any other at- tributes must be numeric. The attribute names in the original data are used to create a string identifier attribute in the transposed data.

**2.3.1.4 String conversion**

A string attribute has an unspecified number of values. *StringToNominal* converts it to nominal with a set number of values. You should ensure that all string values that will appear in potential test data are represented in the dataset. *NominalToString* converts the other way.

*StringToWordVector* produces numeric attributes that represent the frequency of words in the value of a string attribute. The set of words—that is, the new attribute set—is determined from the full set of values in the string attribute. The new attributes can be named with a user-determined prefix to keep attributes derived from different string attributes distinct.

There are many options that affect tokenization. Words can be formed from contiguous alphabetic sequences or separated by a given set of delimiter characters. In the latter case, they

32 CHAPTER 2. THE EXPLORER

can be further split into *n*-grams (with user-supplied minimum and maximum length), or be processed by a stemming algorithm. They can be converted to lowercase before being added to the dictionary, or all words on a supplied list of stopwords can be ignored. Words that are not among the top *k* words ranked by frequency can be discarded (slightly more than *k* words will be retained if there are ties at the *k*th position). If a class attribute has been assigned, the top *k* words for each class will be kept (this can be turned off by the user). The value of each word attribute reflects its presence or absence in the string, but this can be changed. A count of the number of times the word appears in the string can be used instead. Word frequencies can be normalized to give each document’s attribute vector the same Euclidean length—this length is not chosen to be 1, to avoid the very small numbers that would entail, but is the average length of all documents that appear as values of the original string attribute. Alternatively, the frequencies *fij* for word *i* in document *j* can be transformed using log (1 + *fij*) or the TFIDF measure.

*FixedDictionaryStringToWordVector* is similar to *StringToWordVector* but allows the user to provide a dictionary file containing the words that will form the attributes, rather than building a dictionary on the fly. The format of the dictionary file is that of one word per line, with each followed by an optional count (separated by a comma) of how many documents in the corpus from which the dictionary was derived contained that word.

*ChangeDateFormat* alters the formatting string that is used to parse date attributes. Any format supported by Java’s *SimpleDateFormat* class can be specified.

**2.3.1.5 Time series**

Two filters work with time series data. *TimeSeriesTranslate* replaces the values of an attribute (or attributes) in the current instance with the equivalent value in some other (previous or future) instance. *TimeSeriesDelta* replaces attribute values in the current instance with the difference between the current value and the value in some other instance. In both cases instances in which the time-shifted value is unknown may be removed, or missing values used.

**2.3.1.6 Randomizing the attributes**

Other attribute filters degrade the data. *AddNoise* takes a nominal attribute and changes a given percentage of its values. Missing values can be retained or changed along with the rest. *Obfuscate* anonymizes data by renaming the relation, attribute names, and nominal and string attribute values. *RandomProjection* projects the dataset on to a lower-dimensional subspace using a random matrix with columns of unit length. The class attribute is not included in the projection.

**2.3.2 Unsupervised instance filters**

WEKA’s instance filters affect all instances in a dataset rather than all values of a particular attribute or set of attributes.

**2.3.2.1 Randomizing and subsampling**

*Randomize* does just what the name suggests—it randomizes the order of the instances in the dataset. The user can specify a random number seed to use in this process.

There are various ways of generating subsets of the data. Use *Resample* to produce a random sample by sampling with or without replacement, or *RemoveFolds* to split the data into a given number of cross-validation folds and reduce it to just one of them. If a random number seed is

2.3. FILTERING ALGORITHMS 33

provided, the dataset will be shuffled before the subset is extracted. *ReservoirSample* uses the reservoir sampling algorithm to produce a random sample (without replacement) from a dataset. When used from the Knowledge Flow interface (see Chapter 3) or from the command line, the dataset is read incrementally, so that datasets that exceed main memory can be sampled.

*RemovePercentage* removes a given percentage of instances, and *RemoveRange* removes a certain range of instance numbers. To remove all instances that have certain values for nominal attributes, or numeric values above or below a certain threshold, use *RemoveWithValues*. 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. The attribute information is normally left unchanged, but this can be altered so that corresponding values of a nominal attribute are deleted from its definition.

*RemoveFrequentValues* can be used to remove those instances containing the most- or least- frequent values of a particular nominal attribute; the user can specify how many frequent or infrequent values to remove.

*SubsetByExpression* selects all instances that satisfy a user-supplied logical expression. The expression can involve mathematical operators and functions, such as those used by *AddExpres- sion* and *MathExpression*, and logical operators (such as *and*, *or* and *not*), applied to attribute values. For example, the expression

(CLASS is ’mammal’) and (ATT14 *>* 2)

selects only those instances whose class attribute has the value mammal and whose 14th at- tribute exceeds 2.

You can remove outliers by applying a classification method to the dataset (specifying it just as the clustering method was specified previously for *AddCluster*) and use *RemoveMisclassified* to delete the instances that it misclassifies. The process is normally repeated until the data is fully cleansed, but a maximum number of iterations can be specified instead. Cross-validation can be used rather than evaluation on the training data, and for numeric classes an error threshold can be specified.

**2.3.2.2 Sparse instances**

The *NonSparseToSparse* and *SparseToNonSparse* filters convert between the regular representa- tion of a dataset and its sparse representation.

**2.3.3 Supervised filters**

Supervised filters are available from the Explorer’s *Preprocess* panel, just as unsupervised ones are. As mentioned earlier, you need to be careful with them because, despite appearances, they are not really preprocessing operations. We noted this earlier with regard to discretization—it must not use the test data’s class values because these are supposed to be unknown—and this is true for supervised filters in general.

Because of popular demand, WEKA allows you to invoke supervised filters as a preprocessing operation, just like unsupervised filters. However, if you intend to use them for classification you should adopt a different methodology. A *metalearner* is provided so that the process of building the filtering model becomes part of the learning process. This filters the test data using the filter that has been created from the training data. It is also useful for some unsupervised filters. For example, in *StringToWordVector* the dictionary will be created from the training data alone:

34 CHAPTER 2. THE EXPLORER

words that are novel in the test data will be discarded. To use a supervised filter in this way, invoke the *FilteredClassifier* metalearning scheme from the *meta* section of the menu displayed by the Classify panel’s Choose button. Figure 2.17a shows the object editor for this metalearning scheme. With it you choose a classifier and a filter. Figure 2.17b shows the menu of filters.

Supervised filters, just like unsupervised ones, are divided into attribute and instance filters.

**2.3.3.1 Supervised attribute filters**

*Discretize*, highlighted in Figure 2.17, uses the MDL method of supervised discretization. You can specify a range of attributes or force the discretized attribute to be binary. The class must be nominal. By default Fayyad and Irani’s (1993) criterion is used, but Kononenko’s method (1995) is an option.

There is a supervised version of the *NominalToBinary* filter that transforms all multivalued nominal attributes to binary ones. In this version, the transformation depends on whether the class is nominal or numeric. If nominal, the same method as before is used: an attribute with *k* values is transformed into *k* binary attributes. If the class is numeric, however, the method described in used by the model tree learner M5’ is applied. In either case the class itself is not transformed.

*ClassOrder* changes the ordering of the class values. The user determines whether the new ordering is random or in ascending or descending class frequency. This filter must not be used with the *FilteredClassifier* metalearning scheme!

*AddClassification* adds to the data the predictions of a given classifier, which can be either trained on the input dataset or loaded from a file as a serialized object. New attributes can be added that hold the predicted class value, the predicted probability distribution (if the class is nominal), or a flag that indicates misclassified instances (or, for numeric classes, the difference between the predicted and actual class values).

*AttributeSelection* applies a given attribute selection technique in order to reduce the number of attributes in the data. The user can choose which attribute or subset evaluator to use and combine it with a search strategy in order to implement an attribute selection method.

*ClassConditionalProbabilities* converts the values of nominal and numeric attributes into class conditional probability estimates. For each original attribute *A* converted, it creates as many new attributes as there are class values, where the value of the new attribute corresponding to class *Ck* is *P*(*A*|*Ck*). This is essentially the same as what the naive Bayes classifier computes and, in fact, the implementation uses naive Bayes internally. Like the unsupervised filters that merge nominal values, this filter can help when dealing with nominal attributes that have a large number of distinct values.

**2.3.3.2 Supervised instance filters**

There are four supervised instance filters. *Resample* is like the eponymous unsupervised instance filter except that it maintains the class distribution in the subsample. Alternatively, it can be configured to bias the class distribution towards a uniform one. Sampling can be performed with (default) or without replacement. *SpreadSubsample* also produces a random subsample, but the frequency difference between the rarest and the most common class can be controlled—for example, you can specify at most a 2 : 1 difference in class frequencies. You can also limit the number of instances in any class by specifying an explicit maximum count.

*SMOTE* is another filter that samples the data and alters the class distribution (Chawla et al., 2002). Like *SpreadSubsample*, it can be used to adjust the relative frequency between minority and majority classes in the data—but it takes care not to undersample majority classes and it oversamples the minority class by creating synthetic instances using a *k*-nearest neighbor

2.4. LEARNING ALGORITHMS 35

approach. The user can specify the oversampling percentage and the number of neighbors to use when creating synthetic instances.

Like the unsupervised instance filter *RemoveFolds*, *StratifiedRemoveFolds* outputs a specified cross-validation fold for the dataset, except that this time the fold is stratified.

**2.4 Learning algorithms**

On the *Classify* panel, when you select a learning algorithm using the *Choose* button the command-line version of the classifier appears in the line beside the button, including the pa- rameters specified with minus signs. To change them, click that line to get an appropriate object editor. The classifiers in WEKA are divided into Bayesian classifiers, trees, rules, functions, lazy classifiers and a final miscellaneous category. We describe them briefly here, along with their parameters. To learn more, choose one in the WEKA Explorer interface and examine its object editor. A further kind of classifier, the metalearner, is described in the next section.

**2.4.1 Bayesian classifiers**

*NaiveBayes* implements the probabilistic Naive Bayes classifier. *NaiveBayesSimple* uses the normal distribution to model numeric attributes. *NaiveBayes* can use kernel density estimators, which improve performance if the normality assumption is grossly incorrect; it can also handle numeric attributes using supervised discretization.

Figure 2.18 shows the output of *NaiveBayes* on the weather data. The salient difference between this and the output in Figure 2.5 of *J48* on the same data is that instead of a tree in textual form, here the parameters of the *Naive Bayes* model are presented in a table. The first column shows attributes and the other two show class values; entries are either frequency counts of nominal values or parameters of normal distributions for numeric attributes. For example, Figure 2.18 shows that the mean temperature value for instances of class yes is 72.9697, while for instances for which *windy* = *yes* the values *true* and *false* occur 4 and 7 times respectively. The grand total of the yes and no counts for *windy* is, surprisingly, 18—more than the 14 instances in the weather data (the situation for *outlook* is even worse, totaling 20). The reason is that *NaiveBayes* avoids zero frequencies by applying the Laplace correction, which involves initializing each count to 1 rather than to 0.

*NaiveBayesUpdateable* is an incremental version that processes one instance at a time; it can use a kernel estimator but not discretization. *NaiveBayesMultinomial* implements the multino- mial Bayes classifier; *NaiveBayesMultinomialUpdateable* is an incremental version. *NaiveBayes- MultinomialText* is a version of incremental multinomial naive Bayes that can work directly on string attributes. It effectively performs the function of the *StringToWordVector* filter, minus the TF/IDF transformation, on the fly.

*BayesNet* learns Bayesian nets under the assumptions made in Section 6.7: nominal attributes (numeric ones are prediscretized) and no missing values (any such values are replaced globally). There are four different algorithms for estimating the conditional probability tables of the net- work. Search is done using the K2 or the TAN algorithms, or more sophisticated methods based on hill-climbing, simulated annealing, tabu search, and genetic algorithms. Optionally search speed can be improved using AD trees. There are also two algorithms that use conditional inde- pendence tests to learn the structure of the network; alternatively, the network structure can be loaded from an XML file. More details on the implementation of Bayesian networks in WEKA can be found in Bouckaert (2004).

You can observe the network structure by right-clicking the history item and selecting Visu- alize graph. Figure 2.19a shows the graph for the nominal version of the weather data, which in

36 CHAPTER 2. THE EXPLORER

fact corresponds to the Naive Bayes result with all probabilities conditioned on the class value. This is because the search algorithm defaults to K2 with the maximum number of parents of a node set to one. Reconfiguring this to three by clicking on K2 in the configuration panel yields the more interesting network in Figure 2.19b. Clicking on a node shows its probability distribution—Figure 2.19c is obtained by clicking on the windy node in Figure 2.19b.

**2.4.2 Trees**

Of the tree classifiers in WEKA, we have already seen how to use J4.8, which reimplements C4.5. To see the options, click the line beside the *Choose* button in Figure 2.4b to bring up the object editor in Figure 2.20. You can build a binary tree instead of one with multiway branches. You can set the confidence threshold for pruning (default 0.25), and the minimum number of instances permissible at a leaf (default 2). Instead of standard C4.5 pruning you can choose reduced-error pruning. The *numFolds* parameter (default 3) determines the size of the pruning set: the data is divided equally into that number of parts and the last one used for pruning. When visualizing the tree it is nice to be able to consult the original data points, which you can do if *saveInstanceData* has been turned on (it is off, or *False*, by default to reduce memory requirements). You can suppress subtree raising, yielding a more efficient algorithm; force the algorithm to use the unpruned tree instead of the pruned one; or use Laplace smoothing for predicted probabilities.

*DecisionStump*, designed for use with the boosting methods described later, builds one-level binary decision trees for datasets with a categorical or numeric class, dealing with missing values by treating them as a separate value and extending a third branch from the stump. Trees built by *RandomTree* consider a given number of random features at each node, performing no pruning. *RandomForest* constructs random forests by bagging ensembles of random trees.

*REPTree* builds a decision or regression tree using information gain/variance reduction and prunes it using reduced-error pruning. Optimized for speed, it only sorts values for numeric attributes once. It deals with missing values by splitting instances into pieces, as C4.5 does. You can set the minimum number of instances per leaf, maximum tree depth (useful when boosting trees), minimum proportion of training set variance for a split (numeric classes only), and number of folds for pruning.

*M5P* is the model tree learner M5 described in Chapter 7 of the book. *LMT* builds logistic model trees. It can deal with binary and multiclass target variables, numeric and nominal attributes, and missing values. When fitting the logistic regression func- tions at a node using the *LogitBoost* algorithm, it uses cross-validation to determine how many iterations to run just once and employs the same number throughout the tree instead of cross- validating at every node. This heuristic (which you can switch off) improves the run time considerably, with little effect on accuracy. Alternatively, you can set the number of boosting iterations to be used throughout the tree manually, or use a fast heuristic based on the Akaike Information Criterion instead of cross-validation. Weight trimming during the boosting process can be used to further improve run time. Normally, it is the misclassification error that cross- validation minimizes, but the mean absolute error of the probability estimates can be chosen instead. The splitting criterion can be based on C4.5’s information gain (the default) or on the LogitBoost residuals, striving to improve the purity of the residuals. LMT employs the minimal cost-complexity pruning mechanism to produce a compact tree structure.

*HoeffdingTree* is an implementation of the incremental decision tree algorithm described in Chapter 13 of the book. It offers options to create splits based on information gain or the Gini index. Predictions at the leaves of the tree can be made by either majority class or naive Bayes models.

2.4. LEARNING ALGORITHMS 37

**2.4.3 Rules**

*DecisionTable* builds a decision table classifier. It evaluates feature subsets using best-first search and can use cross-validation for evaluation (Kohavi 1995b). An option uses the nearest-neighbor method to determine the class for each instance that is not covered by a decision table entry, instead of the table’s global majority, based on the same set of features.

*OneR* is the 1R classifier with one parameter: the minimum bucket size for discretization. Figure 2.21 shows its output for the labor negotiations data. The Classifier model part shows that wage-increase-first-year has been identified as the basis of the rule produced, with a split at the value 2.9 dividing bad outcomes from good ones (the class is also good if the value of that attribute is missing). Beneath the rules the fraction of training instances correctly classified by the rules is given in parentheses.

*Part* obtains rules from partial decision trees. It builds the tree using C4.5’s heuristics with the same user-defined parameters as J4.8. Figure 2.22 shows the output of Part for the labor negotiations data. Three rules are found, and are intended to be processed in order, the prediction generated for any test instance being the outcome of the first rule that fires. The last, “catch-all,” rule will always fire if no other rule fires. As with J48, the numbers in parentheses that follow each rule give the number of instances that are covered by the rule followed by the number that are misclassified (if any).

*M5Rules* obtains regression rules from model trees built using M5’. *Ridor* learns rules with exceptions by generating the default rule, using incremental reduced-error pruning to find excep- tions with the smallest error rate, finding the best exceptions for each exception, and iterating. *JRip* implements Ripper algorithm, including heuristic global optimization of the rule set (Co- hen 1995). *NNge* is a nearest-neighbor method for generating rules using nonnested generalized exemplars.

**2.4.4 Functions**

Algorithms that fall into the *functions* category include an assorted group of classifiers that can be written down as mathematical equations in a reasonably natural way. Other methods, such as decision trees and rules, cannot (there are exceptions: Naive Bayes has a simple mathematical formulation). Four of them implement linear regression. *SimpleLinearRegression* learns a linear regression model based on a single attribute—it chooses the one that yields the smallest squared error. Missing values and nonnumeric attributes are not allowed. Figure 2.23 shows the output of *SimpleLinearRegression* for the CPU performance data. The attribute that has the smallest squared error in this case is *MMAX*.

*LinearRegression* performs least-squares multiple linear regression including attribute selec- tion, either greedily using backward elimination or by building a full model from all attributes and dropping terms one by one in decreasing order of their standardized coefficients until a stop- ping criteria is reached. Both methods use a version of the AIC termination criterion. Attribute selection can be turned off. The implementation has two further refinements: a heuristic mech- anism for detecting collinear attributes (which can be turned off) and a ridge parameter that stabilizes degenerate cases and can reduce overfitting by penalizing large coefficients. Technically, *LinearRegression* implements ridge regression.

*SMO* implements the sequential minimal optimization algorithm for training a support vector classifier (Platt, 1998; Keerthi et al., 2001), using kernel functions such as polynomial or Gaussian kernels. Missing values are replaced globally, nominal attributes are transformed into binary ones, and attributes are normalized by default—note that the coefficients in the output are based on the normalized data. Normalization can be turned off, or the input standardized to zero mean and unit variance. Pairwise classification is used for multiclass problems. Logistic regression

38 CHAPTER 2. THE EXPLORER

models can be fitted to the support vector machine output to obtain probability estimates. In the multiclass case the predicted probabilities will be combined using pairwise coupling (Hastie and Tibshirani, 1998). When working with sparse instances, turn normalization off for faster operation.

Figure 2.24 shows the output of *SMO* on the iris data. A polynomial kernel with an ex- ponent of 1 has been used, making the model a linear support vector machine. Since the iris data contains three class values, three binary SMO models have been output—one hyperplane to separate each possible pair of class values. Furthermore, since the machine is linear, the hy- perplanes are expressed as functions of the attribute values in the original (albeit normalized) space. Figures 2.25 and 2.26 show the result when the exponent of the polynomial kernel is set to 2, making the support vector machine nonlinear. There are three binary SMO models as before, but this time the classifiers are expressed as functions of the support vectors. Each support vector is shown enclosed in angle brackets, along with the value of its coefficient *α*. The value of the offset parameter, *b*, is shown as the last component of each function.

*SMOreg* implements the sequential minimal optimization algorithm for learning a support vector regression model (Smola and SchÃűlkopf, 1998).

*VotedPerceptron* is the voted perceptron algorithm. *Winnow* modifies the basic perceptron to use multiplicative updates. The implementation allows for a second multiplier *β*—different from 1*/α*—to be used, which makes the algorithm slightly more flexible than the version of balanced Winnow discussed in the book.

*GaussianProcesses* implements the Bayesian Gaussian process technique for non-linear re- gression. Users can specify the kernel function, along with a “noise” regularization parameter for controlling the closeness of fit. They can choose to have the training data normalized or standardized before learning the regression. For point estimates, this method is equivalent to kernel ridge regression.

*SGD* implements stochastic gradient descent learning of linear models. Models can be learned in either a batch or incremental fashion. The loss function chosen determines the type of model learned. Available loss functions include the hinge loss (linear support vector classifier), log loss (logistic regression), squared loss (least squares linear regression), epsilon-insensitive loss (support vector regression) and Huber loss (robust regression). *SGDText*, like *NaiveBayesMulti- nomialText*, is a version that is designed for text classification and can operate directly on string attributes. *SGDText* is limited to just the hinge and log loss.

*SimpleLogistic* builds logistic regression models, fitting them using LogitBoost with simple regression functions as base learners and determining how many iterations to perform using cross- validation—which supports automatic attribute selection. *SimpleLogistic* generates a degenerate logistic model tree comprising a single node, and supports the options given earlier for *LMT*.

*Logistic* is an alternative implementation for building and using a multinomial logistic regres- sion model with a ridge estimator to guard against overfitting by penalizing large coefficients, based on work by le Cessie and van Houwelingen (1992). Figure 2.27 shows its output on the iris data. The coefficients of the regression functions are shown in tabular form, one for each class value except for the last class. Given *m* input attributes and *k* classes, the probability predicted for class *j* (with the exception of the last class) is given by

*P*(*j*|*a*1*,a*2*, ..., am*) = *exp*(*wj*0+*wj*1*a*1+*wj*2*a*2+*...*+*wjmam*)

1+∑*k*−1

*j*=1 *exp*(*wj*0+*wj*1*a*1+*wj*2*a*2+*...*+*wjmam*) where *wj*0 is the intercept term for class *j*. The probability of the last class *k* is given by 1 − ∑*k*−1

*j*=1 *P*(*j*|*a*1*,a*2*, ..., am*)

2.4. LEARNING ALGORITHMS 39

Beneath the table of regression coefficients is a second table giving an estimate of the odds ratio for each input attribute and class value. For a given attribute, this value gives an indication of its influence on the class when the values of the other attributes are held fixed.

**2.4.5 Neural networks**

*MultilayerPerceptron* is a neural network that is trained using back propagation. Although listed under functions, it differs from the other schemes because it has its own user interface. If you load up the numeric version of the weather data, invoke *MultilayerPerceptron*, set *GUI* to *True* in its object editor, and run the network by clicking *Start* on the *Classify* panel, the diagram in Figure 2.28 appears in a separate window. This network has three layers: an input layer on the left with one rectangular box for each attribute (colored green); a hidden layer next to it (red) to which all the input nodes are connected; and an output layer at the right (orange). The labels at the far right show the classes that the output nodes represent. Output nodes for numeric classes are automatically converted to unthresholded linear units.

Before clicking *Start* to run the network, you can alter its structure by adding nodes and connections. Nodes can be selected or deselected. All six nodes in the hidden and output layers in Figure 2.28 are deselected, indicated by the gray color of their center. To select a node, simply click on it. This changes the color of its center from gray to bright yellow. To deselect a node, right-click in an empty space. To add a node, ensure that none is selected and left-click anywhere in the panel; the new node will be selected automatically. In Figure 2.28, a new node has been added at the lower center. To connect two nodes, select the start node and then click on the end one. If several start nodes are selected, they are all connected to the end node. If you click in empty space instead, a new node is created as the end node. Notice that connections are directional (although the directions are not shown). The start nodes remain selected; thus you can add an entire hidden layer with just a few clicks, as shown in Figure 2.28b. To remove a node, ensure that no nodes are selected and right-click it; this also removes all connections to it. To remove a single connection, select one node and right-click the node at the other end.

As well as configuring the structure of the network, you can control the learning rate, the momentum, and the number of passes that will be taken through the data, called *epochs*. The network begins to train when you click *Start*, and a running indication of the epoch and the error for that epoch is shown at the lower left of the panel in Figure 2.28. Note that the error is based on a network that changes as the value is computed. For numeric classes the error value depends on whether the class is normalized. The network stops when the specified number of epochs is reached, at which point you can accept the result or increase the desired number of epochs and press *Start* again to continue training.

*MultilayerPerceptron* need not be run through the graphical interface. Several parameters can be set from the object editor to control its operation. If you are using the graphical interface they govern the initial network structure, which you can override interactively. With *autoBuild* set, hidden layers are added and connected up. The default is to have the one hidden layer shown in Figure 2.28a, but without *autoBuild* this would not appear and there would be no connections. The *hiddenLayers* parameter defines what hidden layers are present and how many nodes each one contains. Figure 2.28a is generated by a value of 4 (one hidden layer with four nodes), and although 2.28b was created by adding nodes interactively, it could have been generated by setting *hiddenLayers* to *4,5* (one hidden layer with four nodes and another with five). The value is a comma-separated list of integers; *0* gives no hidden layers. Furthermore, there are predefined values that can be used instead of integers: *i* is the number of attributes, *o* the number of class values, *a* the average of the two, and *t* their sum. The default, *a*, was used to generate Figure 2.28a.

40 CHAPTER 2. THE EXPLORER

The options *learningRate* and *momentum* set values for these parameters, which can be overridden in the graphical interface. A *decay* parameter causes the learning rate to decrease with time: it divides the starting value by the epoch number to obtain the current rate. This sometimes improves performance and may stop the network from diverging. The reset parameter automatically resets the network with a lower learning rate and begins training again if it is diverging from the answer (this option is only available if the graphical interface is not used).

The *trainingTime* parameter sets the number of training epochs. Alternatively, a percentage of the data can be set aside for validation (using *validationSetSize*): then training continues until performance on the validation set starts to deteriorate consistently—or until the specified number of epochs is reached. If the percentage is set to zero, no validation set is used. The *validationThreshold* parameter determines how many consecutive times the validation set error can deteriorate before training is stopped.

The *NominalToBinaryFilter* filter is specified by default in the *MultilayerPerceptron* object editor; turning it off may improve performance on data in which the nominal attributes are really ordinal. The attributes can be normalized (with *normalizeAttributes*), and a numeric class can be normalized too (with *normalizeNumericClass*). Both may improve performance; these options are turned on by default.

**2.4.6 Lazy classifiers**

Lazy learners store the training instances and do no real work until classification time. The simplest lazy learner is the *k*-nearest-neighbor classifier, which is implemented by *IBk*. A variety of different search algorithms can be used to speed up the task of finding the nearest neighbors. A linear search is the default, but other options include kD-trees, ball trees and so-called “cover trees” (Beygelzimer et al., 2006). The distance function used is a parameter of the search method. The default is the same as for *IB1*, that is, the Euclidean distance; other options include Chebyshev, Manhattan and Minkowski distances. The number of nearest neighbors (default *k* = 1) can be specified explicitly in the object editor or determined automatically using leave-one- out cross-validation, subject to an upper limit given by the specified value. Predictions from more than one neighbor can be weighted according to their distance from the test instance, and two different formulas are implemented for converting the distance into a weight. The number of training instances kept by the classifier can be restricted by setting the window size option. As new training instances are added, the oldest ones are removed to maintain the number of training instances at this size *KStar* is a nearest-neighbor method with generalized distance function based on transformations, discussed in Chapter 7 of the book.

*LWL* is a general algorithm for locally weighted learning. It assigns weights using an instance- based method and builds a classifier from the weighted instances. The classifier is selected in *LWL’s* object editor: a good choice is naive Bayes or logistic regression for classification problems and linear regression for regression problems. You can set the number of neighbors used, which determines the kernel bandwidth, and the kernel shape to use for weighting—linear, inverse, or Gaussian. Attribute normalization is turned on by default.

**2.4.7 Miscellaneous classifiers**

The “Misc.” category includes just two classifiers, unless further corresponding WEKA packages have been installed. *SerializedClassifier* loads a model that has been serialized to a file and uses it for prediction. Providing a new training dataset has no effect, because it encapsulates a static model. Similarly, performing cross-validation using *SerializedClassifier* makes little sense. *InputMappedClassifier* wraps a base classifier (or model that has been serialized to a file) and

2.4. LEARNING ALGORITHMS 41

constructs a mapping between the attributes present in the incoming test data and those that were seen when the model was trained. Values for attributes present in the test data but not in the training data are simply ignored. Values for attributes present at training time but not present in the test data receive missing values. Similarly, missing values are used for incoming nominal values not seen during training.

**2.4.8 Metalearning algorithms**

Metalearning algorithms take classifiers and turn them into more powerful learners. One param- eter specifies the base classifier(s); others specify the number of iterations for iterative schemes such as bagging and boosting and an initial seed for the random number generator. We already met *FilteredClassifier* in Section 2.3.3: it runs a classifier on data that has been passed through a filter, which is a parameter. The filter’s own parameters are based exclusively on the training data, which is the appropriate way to apply a supervised filter to test data.

**2.4.8.1 Bagging and randomization**

*Bagging* bags a classifier to reduce variance. This implementation works for both classification and regression, depending on the base learner. In the case of classification, predictions are generated by averaging probability estimates, not by voting. One parameter is the size of the bags as a percentage of the training set. Another is whether to calculate the out-of-bag error, which gives the average error of the ensemble members (Breiman 2001).

*RandomCommittee* is even simpler: it builds an ensemble of base classifiers and averages their predictions. Each one is based on the same data but uses a different random number seed. This only makes sense if the base classifier is randomized; otherwise, all classifiers would be the same. *RandomSubSpace* builds an ensemble of classifiers, each trained using a randomly selected subset of the input attributes. Aside from the number of iterations and random seed to use, it provides a parameter to control the size of the attribute subsets. *RotationForest* implements the rotation forest ensemble learner. Although the classic paper on rotation forests, Rodriguez et al. (2006), uses random subspaces and principal components to create an ensemble of decision trees, WEKA’s implementation allows the base classifier to be any classification or regression scheme. The principal components transformation is performed by WEKA’s filter of the same name. *RotationForest* can be configured to use other projections such as random projections or partial least squares. Other parameters control the size of the subspaces and the number of instances that are input to the projection filter.

**2.4.8.2 Boosting**

*AdaBoostM1* implements the classic boosting algorithm. It can be accelerated by specifying a threshold for weight pruning. AdaBoostM1 resamples if the base classifier cannot handle weighted instances (you can also force resampling anyway).

Whereas *AdaBoostM1* only applies to nominal classes, *AdditiveRegression* enhances the per- formance of a regression learner. There are two parameters: shrinkage, which governs the learning rate, and the maximum number of models to generate. If the latter is infinite, work continues until the error stops decreasing.

*LogitBoost* performs additive logistic regression. Like *AdaBoostM1*, it can be accelerated by specifying a threshold for weight pruning. The appropriate number of iterations can be determined using internal cross-validation; there is a shrinkage parameter that can be tuned to prevent overfitting; and you can choose resampling instead of reweighting.

42 CHAPTER 2. THE EXPLORER

**2.4.8.3 Combining classifiers**

*Vote* provides a baseline method for combining classifiers. The default scheme is to average their probability estimates or numeric predictions, for classification and regression respectively. Other combination schemes are available, for example, using majority voting for classification.

*Stacking* combines classifiers using stacking for both classification and regression problems. You specify the base classifiers, the metalearner, and the number of cross-validation folds.

**2.4.8.4 Cost-sensitive learning**

There is one metalearner, called *CostSensitiveLearning*, for cost-sensitive learning. The cost matrix can be supplied as a parameter or loaded from a file in the directory set by the *onDe- mandDirectory* property, named by the relation name and with the extension *cost*. *CostSen- sitiveClassifier* either reweights training instances according to the total cost assigned to each class or predicts the class with the minimum expected misclassification cost rather than the most likely one.

**2.4.8.5 Optimizing performance**

Five metalearners use the wrapper technique to optimize the base classifier’s performance. *At- tributeSelectedClassifier* selects attributes, reducing the data’s dimensionality before passing it to the classifier. You can choose the attribute evaluator and search method as in the *Select attributes* panel described in Section 2.2. *CVParameterSelection* optimizes performance by using cross-validation to select parameters. For each parameter you give a string containing its lower and upper bounds and the desired number of increments. For example, to vary parameter –P from 1 to 10 in increments of 1, use

P 1 10 10

The number of cross-validation folds to be used can be specified. *MultiScheme* selects a classifier to use from among several by using either the resubstitution error or cross-validation on the training data. Performance is measured using percentage correct in the case of classification and mean squared error for regression.

*IterativeClassifierOptimizer* optimizes the number of iterations performed by iterative clas- sifiers, such as boosting methods, using cross-validation on the training data. Performance can be measured using any of the metrics described in Chapter 5 of the book. The user can specify parameters such as how many cross-validation folds to use, how many iterations to look ahead in order to find a better solution, and how often to perform the evaluation (if evaluation of every iteration is proving too time consuming).

The fifth metalearner, *ThresholdSelector*, optimizes one of a number of different evaluation metrics, including F-measure, precision, recall, accuracy and true positive rate, by selecting a probability threshold on the classifier’s output. Performance can measured on the training data, on a holdout set, or by cross-validation. The probabilities returned by the base learner can be rescaled into the full range [0,1], which is useful if the scheme’s probabilities are restricted to a narrow subrange. The metalearner can be applied to multiclass problems by specifying the class value for which the optimization is performed as

1. The first class value.

2. The second class value.

2.5. CLUSTERING ALGORITHMS 43

3. Whichever value is least frequent.

4. Whichever value is most frequent.

5. The first class named *yes*, *pos(itive)*, or *1*.

**2.4.8.6 Retargeting classifiers for different tasks**

Three metalearners adapt learners designed for one kind of task to another. *ClassificationViaRe- gression* performs classification using a regression method by binarizing the class and building a regression model for each value. *RegressionByDiscretization* is a regression scheme that dis- cretizes the class attribute into a specified number of bins using equal-width discretization and then employs a classifier. The predictions are the weighted average of the mean class value for each discretized interval, with weights based on the predicted probabilities for the intervals.

*MultiClassClassifier* handles multiclass problems with two-class classifiers using any of these methods:

1. One versus all the rest.

2. Pairwise classification using voting to predict.

3. Exhaustive error-correcting codes.

4. Randomly selected error-correcting codes.

Random code vectors are known to have good error-correcting properties: a parameter spec- ifies the length of the code vector as a factor that is multiplied by the number of classes in the data. For pairwise classification, pairwise coupling of probability estimates can be turned on.

**2.5 Clustering algorithms**

The *Cluster* panel provides access to clustering algorithms. *Cobweb* and *SimpleKMeans* are de- scribed in Chapter 4 of the book; *EM* is described in Chapter 9. For the *EM* implementation you can specify how many clusters to generate, or the algorithm can decide by cross-validating the loglikelihood—in which case the number of folds is fixed at 10 (unless there are fewer than 10 training instances). You can specify the maximum number of iterations and set the min- imum allowable standard deviation for the normal density calculation. Clusters are Gaussian distributions with diagonal covariance matrices. *SimpleKMeans* clusters data using *k*-means; the number of clusters is specified by a parameter. The user can choose between the Euclidean and Manhattan distance metrics. In the latter case the algorithm is actually *k*-medians instead *k*-means, and the centroids are based on medians rather than means in order to minimize the within-cluster distance function.

Figure 2.29 shows the output of *SimpleKMeans* for the weather data, with default options: two clusters and Euclidean distance. The result of clustering is shown as a table whose rows are attribute names and whose columns correspond to the cluster centroids; an additional cluster at the beginning shows the entire data set. The number of instances in each cluster appears in parentheses at the top of its column. Each table entry is either the mean (numeric attribute) or mode (nominal attribute) of the corresponding attribute for the cluster in that column; users can choose to show standard deviations (numeric attribute) and frequency counts (nominal at- tributes) as well. The bottom of the output shows the result of applying the learned clustering model. In this case, it assigned each training instance to one of the clusters, showing the same

44 CHAPTER 2. THE EXPLORER

result as the parenthetical numbers at the top of each column. An alternative is to use a separate test set or a percentage split of the training data, in which case the figures would be different.

Figure 2.29 shows the output of *EM* for the same data, with the number of clusters set to two. Although there is no notion of the number of instances in each cluster, the columns are again headed by its prior probability in parentheses. The table entries show the parameters of normal distributions for numeric attributes or frequency counts for the values of nominal attributes—and here the fractional count values reveal the “soft” nature of the clusters produced by the EM algorithm, in that any instance can be split between several clusters. At the bottom the loglikelihood of the model (again with respect to the training data) is shown, as well as the number of instances assigned to each cluster when the learned model is applied to the data as a classifier.

*Cobweb* implements both the Cobweb algorithm for nominal attributes and the Classit algo- rithm for numeric attributes. The ordering and priority of the merging and splitting operators differs from the original Cobweb and Classit papers (where it is somewhat ambiguous). This implementation always compares four different ways of treating a new instance and chooses the best: adding it to the best host, making it into a new leaf, merging the two best hosts and adding it to the merged node, and splitting the best host and adding it to one of the splits. *Acuity* and *cutoff* are parameters.

*HierarchicalClusterer* implements agglomerative (bottom-up) generation of hierarchical clus- ters (Section 6.8 of the book). Several different link types, which are ways of measuring the distance between clusters, are available as options.

*FarthestFirst* implements the farthest-first traversal algorithm of Hochbaum and Shmoys (1985), cited by Sanjoy Dasgupta (2002); a fast, simple, approximate clusterer modeled on *k*- means. *MakeDensityBasedClusterer* is a metaclusterer that wraps a clustering algorithm to make it return a probability distribution and density. To each cluster and attribute it fits a discrete distribution or a symmetric normal distribution (whose minimum standard deviation is a parameter).

*Canopy* implements the canopy clustering algorithm algorithm of McCallum, Nigam and Ungar (2000). Canopy clustering is often used as an initialization strategy for *k*-means or as a fast approximate clustering method for use on large datasets, where applying another algorithm directly might be impractical. The algorithm partitions the input data into proximity regions (canopies) in the form of hyperspheres defined by a loose distance, *T1*, from the region’s center. A second tight distance, *T2* (where *T*2 *< T*1), is used to control how many canopies are formed by the algorithm. At most two passes over the data are required. The first pass creates the canopies, and it starts by choosing one instance at random to form the center of the first canopy. Following this, each of the remaining instances are considered in turn, by computing its distance to the center of each existing canopy. If a given instance is outside the T2 distance to all canopies then it forms a new canopy, otherwise it is discarded. The second pass over the data assigns instances to canopies and makes use of the loose *T1* distance. Because canopies can overlap according to *T1* distance, it is possible for a given instance to belong to more than one canopy.

**2.6 Association rule learners**

WEKA has three association rule learners. *Apriori* implements the Apriori algorithm. It starts with a minimum support of 100% of the data items and decreases this in steps of 5% until there are at least 10 rules with the required minimum confidence of 0.9 or until the support has reached a lower bound of 10%, whichever occurs first. (These default values can be changed.) There are four alternative metrics for ranking rules: *Confidence*, which is the proportion of

2.7. ATTRIBUTE SELECTION 45

the examples covered by the premise that are also covered by the consequent; *Lift*, which is determined by dividing the confidence by the support; *Leverage*, which is the proportion of additional examples covered by both the premise and the consequent beyond those expected if the premise and consequent were statistically independent; and *Conviction*, a measure defined by Brin et al. (1997). You can also specify a significance level, and rules will be tested for significance at this level. Apriori has an option to limit the rules found to those that contain just the value of a single attribute in the consequence of the rule. Such rules are called “class” association rules—i.e., classification rules.

In order to process market-basket data with Apriori, where we are interested in knowing (from the items present in shoppers’ baskets) which items are purchased together, it is necessary to encode the input ARFF data in a specific way. In particular, since we are not interested in co-occurrence of items not present in shopping baskets, the attributes corresponding to items should be declared as single-valued nominal ones in the ARFF file. Missing values can be used to indicate the absence of an item from a shopping basket.

*FPGrowth* implements the frequent pattern tree mining algorithm. Being designed for market- basket data, Apriori’s special encoding for this type of data is not implemented here. All at- tributes are expected to be binary nominal ones, and the user can specify which of the two values is to be treated as positive, i.e., indicates presence in the basket (the default is to use the second value). FPGrowth can operate on either standard or sparse instances. Most of its options are the same as for Apriori. It finds the requested number of rules in the same way—by iteratively decreasing the minimum support. Optionally, the user can have FPGrowth find all the rules that meet the lower bound for the minimum support and the minimum value set for the ranking metric (confidence, lift, leverage or conviction).

*FilteredAssociator* allows the data to be passed through a filter before it reaches an associator. Both the filter and the base associator are options that the user can configure.

**2.7 Attribute selection**

Figure 2.31 shows that part of WEKA’s attribute selection panel where you specify the attribute evaluator and search method. Attribute selection is normally done by searching the space of attribute subsets, evaluating each one. A potentially faster but less accurate approach is to evaluate the attributes individually and sort them, discarding attributes that fall below a chosen cut-off point.

**2.7.1 Attribute subset evaluators**

Subset evaluators take a subset of attributes and return a numerical measure that guides the search. They are configured like any other WEKA object. *CfsSubsetEval* assesses the predictive ability of each attribute individually and the degree of redundancy among them, preferring sets of attributes that are highly correlated with the class but with low intercorrelation. An option iteratively adds attributes that have the highest correlation with the class, provided that the set does not already contain an attribute whose correlation with the attribute in question is even higher. Missing can be treated as a separate value, or its counts can be distributed among other values in proportion to their frequency.

Whereas *CfsSubsetEval* is a filter method of attribute selection, *WrapperSubsetEval* imple- ments a wrapper method. *WrapperSubsetEval* uses a classifier to evaluate attribute set and employs cross-validation to estimate the accuracy of the learning scheme for each set.

46 CHAPTER 2. THE EXPLORER

**2.7.2 Single-attribute evaluators**

Single-attribute evaluators are used with the *Ranker* search method to generate a ranked list from which *Ranker* discards a given number (explained in the next subsection). *ReliefFAttributeEval* is instance-based: it samples instances randomly and checks neighboring instances of the same and different classes. It operates on discrete and continuous class data. Parameters specify the number of instances to sample, the number of neighbors to check, whether to weight neighbors by distance, and an exponential function that governs how rapidly weights decay with distance. *InfoGainAttributeEval* evaluates attributes by measuring their information gain with respect to the class. It discretizes numeric attributes first using the MDL-based discretization method (it can be set to binarize them instead). This method, along with the next three, can treat missing as a separate value or distribute the counts among other values in proportion to their frequency. *ChiSquaredAttributeEval* evaluates attributes by computing the chi-squared statistic with respect to the class. GainRatioAttributeEval evaluates attributes by measuring their gain ratio with respect to the class. *SymmetricalUncertAttributeEval* evaluates an attribute by measuring its symmetrical uncertainty with respect to the class.

*OneRAttributeEval* uses the simple accuracy measure adopted by the *OneR* classifier. It can use the training data for evaluation, as OneR does, or it can apply internal cross-validation: the number of folds is a parameter. It adopts *OneR’s* simple discretization method: the minimum bucket size is a parameter.

Unlike other single-attribute evaluators, *PrincipalComponents* transforms the set of attributes. The new attributes are ranked in order of their eigenvalues. Optionally, a subset is selected by choosing sufficient eigenvectors to account for a given proportion of the variance (95% by default). Finally, the reduced data can be transformed back to the original space.

**2.7.3 Search methods**

Search methods traverse the attribute space to find a good subset. Quality is measured by the chosen attribute subset evaluator. Each search method can be configured with WEKA’s object editor. *BestFirst* performs greedy hill climbing with backtracking; you can specify how many consecutive nonimproving nodes must be encountered before the system backtracks. It can search forward from the empty set of attributes, backward from the full set, or start at an intermediate point (specified by a list of attribute indexes) and search in both directions by considering all possible single-attribute additions and deletions. Subsets that have been evaluated are cached for efficiency; the cache size is a parameter.

*GreedyStepwise* searches greedily through the space of attribute subsets. Like *BestFirst*, it may progress forward from the empty set or backward from the full set. Unlike *BestFirst*, it does not backtrack but terminates as soon as adding or deleting the best remaining attribute decreases the evaluation metric. In an alternative mode, it ranks attributes by traversing the space from empty to full (or vice versa) and recording the order in which attributes are selected. You can specify the number of attributes to retain or set a threshold below which attributes are discarded.

Finally we describe *Ranker*, which as noted earlier is not a search method for attribute subsets but a ranking scheme for individual attributes. It sorts attributes by their individual evaluations and must be used in conjunction with one of the single-attribute evaluators—not an attribute subset evaluator. *Ranker* not only ranks attributes but also performs attribute selection by removing the lower-ranking ones. You can set a cut-off threshold below which attributes are discarded, or specify how many attributes to retain. You can specify certain attributes that must be retained regardless of their rank.

2.7. ATTRIBUTE SELECTION 47

(b) *J4.8*’s errors on the iris data.

Figure 2.6: Visualizing the result of *J4.8* on the iris data.

(a) *J4.8* tree for the iris data.

48 CHAPTER 2. THE EXPLORER

(a) Generic Object Editor.

(b) More information.

(c) Choosing a converter (click Choose).

Figure 2.7: The Generic Object Editor.

2.7. ATTRIBUTE SELECTION 49

Figure 2.8: The SQLViewer tool.

50 CHAPTER 2. THE EXPLORER

(d) Information about the filter’s capabilities (click *Capabilities*).

(b) An object editor.

(e) Constraints on capabilities.

(c) More information (click *More*).

Figure 2.9: Choosing a filter.

(a) The filters menu.

2.7. ATTRIBUTE SELECTION 51

Figure 2.10: The weather data with two attributes removed.

52 CHAPTER 2. THE EXPLORER

(b) *M5’*’s performance on the CPU data.

Figure 2.11: Processing the CPU performance data with *M5’*.

(a) CPU data in the *Preprocess* panel.

2.7. ATTRIBUTE SELECTION 53

=== Run information ===

Scheme: weka.classifiers.trees.M5P -M 4.0 Relation: cpu Instances: 209 Attributes: 7MYCT MMIN MMAX CACH CHMIN CHMAX class Test mode: 10-fold cross-validation

=== Classifier model (full training set) ===

M5 pruned model tree: (using smoothed linear models)

CHMIN <= 7.5 : LM1 (165/12.903%) CHMIN > 7.5 : | MMAX <= 28000 : | | MMAX <= 13240 : | | | CACH <= 81.5 : LM2 (6/18.551%) | | | CACH > 81.5 : LM3 (4/30.824%) | | MMAX > 13240 : LM4 (11/24.185%) | MMAX > 28000 : LM5 (23/48.302%)

LM num: 1 class = -0.0055 \* MYCT + 0.0013 \* MMIN + 0.0029 \* MMAX + 0.8007 \* CACH + 0.4015 \* CHMAX

+ 11.0971

LM num: 2 class = -1.0307 \* MYCT + 0.0086 \* MMIN + 0.0031 \* MMAX + 0.7866 \* CACH - 2.4503 \* CHMIN

+ 1.1597 \* CHMAX + 70.8672

LM num: 3 class = -1.1057 \* MYCT + 0.0086 \* MMIN + 0.0031 \* MMAX + 0.7995 \* CACH

- 2.4503 \* CHMIN + 1.1597 \* CHMAX + 83.0016

LM num: 4 class = -0.8813 \* MYCT + 0.0086 \* MMIN + 0.0031 \* MMAX + 0.6547 \* CACH

- 2.3561 \* CHMIN + 1.1597 \* CHMAX + 82.5725

LM num: 5 class = -0.4882 \* MYCT + 0.0218 \* MMIN + 0.003 \* MMAX + 0.3865 \* CACH

- 1.3252 \* CHMIN + 3.3671 \* CHMAX - 51.8474

Number of Rules : 5

Time taken to build model: 0.14 seconds

=== Cross-validation === === Summary ===

Correlation coefficient 0.9274 Mean absolute error 29.8309 Root mean squared error 60.7112 Relative absolute error 30.9967 % Root relative squared error 37.7434 % Total Number of Instances 209

Figure 2.12: Output from the M5’ program for numeric prediction.

54 CHAPTER 2. THE EXPLORER

(b) Linear regression.

Figure 2.13: Visualizing errors.

(a) M5’.

Figure 2.14: Configuring a metalearner for boosting decision stumps.

1. outlook=overcast 4 ==> play=yes 4 <conf:(1)> lift:(1.56) lev:(0.1) [1] conv:(1.43) 2. temperature=cool 4 ==> humidity=normal 4 <conf:(1)> lift:(2) lev:(0.14) [2] conv:(2) 3. humidity=normal windy=FALSE 4 ==> play=yes 4 <conf:(1)> lift:(1.56) lev:(0.1) [1] conv:(1.43) 4. outlook=sunny play=no 3 ==> humidity=high 3 <conf:(1)> lift:(2) lev:(0.11) [1] conv:(1.5) 5. outlook=sunny humidity=high 3 ==> play=no 3 <conf:(1)> lift:(2.8) lev:(0.14) [1] conv:(1.93) 6. outlook=rainy play=yes 3 ==> windy=FALSE 3 <conf:(1)> lift:(1.75) lev:(0.09) [1] conv:(1.29) 7. outlook=rainy windy=FALSE 3 ==> play=yes 3 <conf:(1)> lift:(1.56) lev:(0.08) [1] conv:(1.07) 8. temperature=cool play=yes 3 ==> humidity=normal 3 <conf:(1)> lift:(2) lev:(0.11) [1] conv:(1.5) 9. outlook=sunny temperature=hot 2 ==> humidity=high 2 <conf:(1)> lift:(2) lev:(0.07) [1] conv:(1) 10. temperature=hot play=no 2 ==> outlook=sunny 2 <conf:(1)> lift:(2.8) lev:(0.09) [1] conv:(1.29)

Figure 2.15: Output from the Apriori program for association rules.

2.7. ATTRIBUTE SELECTION 55

56 CHAPTER 2. THE EXPLORER

(a) Scatter plot matrix.

2.7. ATTRIBUTE SELECTION 57

(b) (a) Configuring *FilteredClassifier*.

The menu of filters.

Figure 2.17: Using WEKA’s metalearner for discretization.

58 CHAPTER 2. THE EXPLORER

=== Run information ===

Scheme: weka.classifiers.bayes.NaiveBayes Relation: weather Instances: 14 Attributes: 5outlook

temperature humidity windy play Test mode: 10-fold cross-validation

=== Classifier model (full training set) ===

Naive Bayes Classifier

Class Attribute yes no (0.63) (0.38) =============================== outlook

sunny 3.0 4.0 overcast 5.0 1.0 rainy 4.0 3.0 [total] 12.0 8.0

temperature

mean 72.9697 74.8364 std. dev. 5.2304 7.384 weight sum 9 5 precision 1.9091 1.9091

humidity

mean 78.8395 86.1111 std. dev. 9.8023 9.2424 weight sum 9 5 precision 3.4444 3.4444

windy

TRUE 4.0 4.0 FALSE 7.0 3.0 [total] 11.0 7.0

Time taken to build model: 0 seconds

=== Stratified cross-validation === === Summary ===

Correctly Classified Instances 9 64.2857 % Incorrectly Classified Instances 5 35.7143 % Kappa statistic 0.1026 Mean absolute error 0.4649 Root mean squared error 0.543 Relative absolute error 97.6254 % Root relative squared error 110.051 % Total Number of Instances 14

=== Detailed Accuracy By Class ===

TP Rate FP Rate Precision Recall F-Measure MCC ROC Area PRC Area Class 0.889 0.800 0.667 0.889 0.762 0.122 0.444 0.633 yes 0.200 0.111 0.500 0.200 0.286 0.122 0.444 0.397 no Weighted Avg. 0.643 0.554 0.607 0.643 0.592 0.122 0.444 0.548

=== Confusion Matrix ===

a b <-- classified as 8 1 | a = yes 4 1 | b = no

Figure 2.18: Output of *NaiveBayes* for the weather data.

2.7. ATTRIBUTE SELECTION 59

(c) The probability distribution table for the windy node in (b).

Figure 2.19: Visualizing a Bayesian network for the weather data (nominal version).

(a) Default outpuit.

(b) Maximum number of parents set to three in the search algorithm.

60 CHAPTER 2. THE EXPLORER

Figure 2.20: Changing the parameters for J4.8.

2.7. ATTRIBUTE SELECTION 61

=== Run information ===

Scheme: weka.classifiers.rules.OneR -B 6 Relation: labor-neg-data Instances: 57 Attributes: 17duration

wage-increase-first-year wage-increase-second-year wage-increase-third-year cost-of-living-adjustment working-hours pension standby-pay shift-differential education-allowance statutory-holidays vacation longterm-disability-assistance contribution-to-dental-plan bereavement-assistance contribution-to-health-plan class Test mode: 10-fold cross-validation

=== Classifier model (full training set) ===

wage-increase-first-year: < 2.9-> bad >= 2.9-> good ?-> good (48/57 instances correct)

Time taken to build model: 0 seconds

=== Stratified cross-validation === === Summary ===

Correctly Classified Instances 41 71.9298 % Incorrectly Classified Instances 16 28.0702 % Kappa statistic 0.3382 Mean absolute error 0.2807 Root mean squared error 0.5298 Relative absolute error 61.3628 % Root relative squared error 110.9627 % Total Number of Instances 57

=== Detailed Accuracy By Class ===

TP Rate FP Rate Precision Recall F-Measure MCC ROC Area PRC Area Class 0.450 0.135 0.643 0.450 0.529 0.349 0.657 0.482 bad 0.865 0.550 0.744 0.865 0.800 0.349 0.657 0.731 good Weighted Avg. 0.719 0.404 0.709 0.719 0.705 0.349 0.657 0.644

=== Confusion Matrix ===

a b <-- classified as 9 11 | a = bad 5 32 | b = good

Figure 2.21: Output of *OneR* for the labor negotiations data.

62 CHAPTER 2. THE EXPLORER

=== Run information ===

Scheme: weka.classifiers.rules.PART -M 2 -C 0.25 -Q 1 Relation: labor-neg-data Instances: 57 Attributes: 17duration

wage-increase-first-year wage-increase-second-year wage-increase-third-year cost-of-living-adjustment working-hours pension standby-pay shift-differential education-allowance statutory-holidays vacation longterm-disability-assistance contribution-to-dental-plan bereavement-assistance contribution-to-health-plan class Test mode: 10-fold cross-validation

=== Classifier model (full training set) ===

PART decision list ------------------

wage-increase-first-year > 2.5 AND longterm-disability-assistance = yes AND statutory-holidays > 10: good (25.67)

wage-increase-first-year <= 4 AND working-hours > 36: bad (19.4/1.58)

: good (11.93/2.18)

Number of Rules : 3

Time taken to build model: 0.01 seconds

=== Stratified cross-validation === === Summary ===

Correctly Classified Instances 45 78.9474 % Incorrectly Classified Instances 12 21.0526 % Kappa statistic 0.5378 Mean absolute error 0.2884 Root mean squared error 0.4339 Relative absolute error 63.0507 % Root relative squared error 90.8836 % Total Number of Instances 57

=== Detailed Accuracy By Class ===

TP Rate FP Rate Precision Recall F-Measure MCC ROC Area PRC Area Class 0.700 0.162 0.700 0.700 0.700 0.538 0.726 0.613 bad 0.838 0.300 0.838 0.838 0.838 0.538 0.726 0.758 good Weighted Avg. 0.789 0.252 0.789 0.789 0.789 0.538 0.726 0.707

=== Confusion Matrix ===

a b <-- classified as 14 6 | a = bad 6 31 | b = goodFigure 2.22: Output of *Part* for the labor negotiations data.

2.7. ATTRIBUTE SELECTION 63

=== Run information ===

Scheme: weka.classifiers.functions.SimpleLinearRegression Relation: cpu Instances: 209 Attributes: 7MYCT MMIN MMAX CACH CHMIN CHMAX class Test mode: 10-fold cross-validation

=== Classifier model (full training set) ===

Linear regression on MMAX

0.01 \* MMAX - 34

Predicting 0 if attribute value is missing.

Time taken to build model: 0 seconds

=== Cross-validation === === Summary ===

Correlation coefficient 0.7844 Mean absolute error 53.8054 Root mean squared error 99.5674 Relative absolute error 55.908 % Root relative squared error 61.8997 % Total Number of Instances 209

Figure 2.23: Output of *SimpleLinearRegression* on the CPU performance data.

64 CHAPTER 2. THE EXPLORER

=== Run information ===

Scheme: weka.classifiers.functions.SMO -C 1.0 -L 0.001 -P 1.0E-12 -N 0 -V -1 -W 1

-K ‘‘weka.classifiers.functions.supportVector.PolyKernel -E 1.0 -C 250007’’ -calibrator ‘‘weka.classifiers.functions.Logistic -R 1.0E-8 -M -1 -num-decimal-places 4’’ Relation: iris Instances: 150 Attributes: 5sepallength

sepalwidth petallength petalwidth class Test mode: 10-fold cross-validation

=== Classifier model (full training set) ===

SMO

Kernel used:

Linear Kernel: K(x,y) = <x,y>

Classifier for classes: Iris-setosa, Iris-versicolor

BinarySMO Machine linear: showing attribute weights, not support vectors.

0.6829 \* (normalized) sepallength + -1.523 \* (normalized) sepalwidth + 2.2034 \* (normalized) petallength + 1.9272 \* (normalized) petalwidth - 0.7091

Number of kernel evaluations: 352 (70.32% cached)

Classifier for classes: Iris-setosa, Iris-virginica

BinarySMO Machine linear: showing attribute weights, not support vectors.

0.5886 \* (normalized) sepallength + -0.5782 \* (normalized) sepalwidth + 1.6429 \* (normalized) petallength + 1.4777 \* (normalized) petalwidth - 1.1668

Number of kernel evaluations: 284 (68.996% cached)

Classifier for classes: Iris-versicolor, Iris-virginica

BinarySMO Machine linear: showing attribute weights, not support vectors.

0.3176 \* (normalized) sepallength + -0.863 \* (normalized) sepalwidth + 3.0543 \* (normalized) petallength + 4.0815 \* (normalized) petalwidth - 4.5924

Number of kernel evaluations: 453 (61.381% cached)

Time taken to build model: 0.03 seconds

=== Stratified cross-validation === === Summary ===

Correctly Classified Instances 144 96 % Incorrectly Classified Instances 6 4 % Kappa statistic 0.94 Mean absolute error 0.2311 Root mean squared error 0.288 Relative absolute error 52 % Root relative squared error 61.101 % Total Number of Instances 150

=== Detailed Accuracy By Class ===

TP Rate FP Rate Precision Recall F-Measure MCC ROC Area PRC Area Class 1.000 0.000 1.000 1.000 1.000 1.000 1.000 1.000 Iris-setosa 0.980 0.050 0.907 0.980 0.942 0.913 0.965 0.896 Iris-versicolor 0.900 0.010 0.978 0.900 0.938 0.910 0.970 0.930 Iris-virginica Weighted Avg. 0.960 0.020 0.962 0.960 0.960 0.941 0.978 0.942

=== Confusion Matrix ===

a b c <-- classified as 50 0 0 | a = Iris-setosa

0 49 1 | b = Iris-versicolor 0 5 45 | c = Iris-virginica

Figure 2.24: Output of *SMO* on the iris data.

2.7. ATTRIBUTE SELECTION 65

=== Run information ===

Scheme: weka.classifiers.functions.SMO -C 1.0 -L 0.001 -P 1.0E-12 -N 0 -V -1 -W 1

-K ‘‘weka.classifiers.functions.supportVector.PolyKernel -E 2.0 -C 250007’’ -calibrator ‘‘weka.classifiers.functions.Logistic -R 1.0E-8 -M -1 -num-decimal-places 4’’ Relation: iris Instances: 150 Attributes: 5sepallength

sepalwidth petallength petalwidth class Test mode: 10-fold cross-validation

=== Classifier model (full training set) ===

SMO

Kernel used:

Poly Kernel: K(x,y) = <x,y>ˆ2.0

Classifier for classes: Iris-setosa, Iris-versicolor

BinarySMO

1 \* <0.333333 0.166667 0.457627 0.375 > \* X] - 1 \* <0.222222 0.541667 0.118644 0.166667 > \* X] - 1 \* <0.138889 0.416667 0.067797 0.083333 > \* X] - 1 \* <0.166667 0.416667 0.067797 0.041667 > \* X] + 1 \* <0.222222 0.208333 0.338983 0.416667 > \* X] - 1 \* <0.055556 0.125 0.050847 0.083333 > \* X] - 1 \* <0.027778 0.375 0.067797 0.041667 > \* X] + 1 \* <0.166667 0.166667 0.389831 0.375 > \* X] + 1 \* <0.361111 0.208333 0.491525 0.416667 > \* X] + 1 \* <0.194444 0 0.423729 0.375 > \* X] - 1 \* <0.194444 0.416667 0.101695 0.041667 > \* X] - 1 \* <0.138889 0.458333 0.101695 0.041667 > \* X] + 1 \* <0.194444 0.125 0.389831 0.375 > \* X] + 0.3697 \* <0.361111 0.375 0.440678 0.5 > \* X] - 0.4599 \* <0.138889 0.416667 0.067797 0 > \* X] - 0.9098 \* <0.194444 0.625 0.101695 0.208333 > \* X] + 1 \* <0.333333 0.166667 0.474576 0.416667 > \* X] + 1 \* <0.388889 0.25 0.423729 0.375 > \* X] - 0.8085

Number of support vectors: 18

Number of kernel evaluations: 2416 (72.564% cached)

Classifier for classes: Iris-setosa, Iris-virginica

BinarySMO

1 \* <0.166667 0.208333 0.59322 0.666667 > \* X] - 0.856 \* <0.055556 0.125 0.050847 0.083333 > \* X] + 0.1315 \* <0.555556 0.333333 0.694915 0.583333 > \* X] - 1 \* <0.222222 0.541667 0.118644 0.166667 > \* X] + 1 \* <0.472222 0.083333 0.677966 0.583333 > \* X] - 0.2756 \* <0.194444 0.625 0.101695 0.208333 > \* X] - 1.0183

Number of support vectors: 6

Number of kernel evaluations: 1364 (60.726% cached)

Classifier for classes: Iris-versicolor, Iris-virginica

BinarySMO

1 \* <0.555556 0.208333 0.677966 0.75 > \* X] - 1 \* <0.305556 0.416667 0.59322 0.583333 > \* X] - 1 \* <0.666667 0.458333 0.627119 0.583333 > \* X] - 1 \* <0.472222 0.583333 0.59322 0.625 > \* X] + 1 \* <0.444444 0.416667 0.694915 0.708333 > \* X] - 1 \* <0.527778 0.083333 0.59322 0.583333 > \* X] + 1 \* <0.416667 0.291667 0.694915 0.75 > \* X] - 1 \* <0.472222 0.291667 0.694915 0.625 > \* X] + 0.4559 \* <0.555556 0.375 0.779661 0.708333 > \* X] - 1 \* <0.666667 0.416667 0.677966 0.666667 > \* X] + 1 \* <0.611111 0.416667 0.762712 0.708333 > \* X] - 1 \* <0.5 0.375 0.627119 0.541667 > \* X] - 1 \* <0.722222 0.458333 0.661017 0.583333 > \* X] + 1 \* <0.472222 0.083333 0.677966 0.583333 > \* X] + 1 \* <0.583333 0.458333 0.762712 0.708333 > \* X] + 1 \* <0.611111 0.5 0.694915 0.791667 > \* X] + 1 \* <0.5 0.416667 0.661017 0.708333 > \* X] - 1 \* <0.694444 0.333333 0.644068 0.541667 > \* X] - 1 \* <0.5 0.416667 0.610169 0.541667 > \* X] + 1 \* <0.416667 0.291667 0.694915 0.75 > \* X] + 1 \* <0.527778 0.333333 0.644068 0.708333 > \* X] - 1 \* <0.444444 0.5 0.644068 0.708333 > \* X] + 1 \* <0.5 0.25 0.779661 0.541667 > \* X] + 1 \* <0.555556 0.291667 0.661017 0.708333 > \* X] + 1 \* <0.361111 0.333333 0.661017 0.791667 > \* X] - 1 \* <0.555556 0.208333 0.661017 0.583333 > \* X] - 0.4559 \* <0.555556 0.125 0.576271 0.5 > \* X] + 1 \* <0.555556 0.333333 0.694915 0.583333 > \* X] + 1 \* <0.166667 0.208333 0.59322 0.666667 > \* X] + 1 \* <0.805556 0.416667 0.813559 0.625 > \* X] - 1 \* <0.555556 0.541667 0.627119 0.625 > \* X] + 1 \* <0.472222 0.416667 0.644068 0.708333 > \* X] - 1 \* <0.361111 0.416667 0.59322 0.583333 > \* X] - 1 \* <0.583333 0.5 0.59322 0.583333 > \* X] - 1 \* <0.472222 0.375 0.59322 0.583333 > \* X] - 1 \* <0.611111 0.333333 0.610169 0.583333 > \* X] - 3.5378

Number of support vectors: 36

Number of kernel evaluations: 3524 (66.711% cached)

Time taken to build model: 0.01 seconds

Figure 2.25: Output of *SMO* with a nonlinear kernel on the iris data.

66 CHAPTER 2. THE EXPLORER

=== Stratified cross-validation === === Summary ===

Correctly Classified Instances 144 96 % Incorrectly Classified Instances 6 4 % Kappa statistic 0.94 Mean absolute error 0.2311 Root mean squared error 0.288 Relative absolute error 52 % Root relative squared error 61.101 % Total Number of Instances 150

=== Detailed Accuracy By Class ===

TP Rate FP Rate Precision Recall F-Measure MCC ROC Area PRC Area Class 1.000 0.000 1.000 1.000 1.000 1.000 1.000 1.000 Iris-setosa 0.940 0.030 0.940 0.940 0.940 0.910 0.955 0.904 Iris-versicolor 0.940 0.030 0.940 0.940 0.940 0.910 0.972 0.916 Iris-virginica Weighted Avg. 0.960 0.020 0.960 0.960 0.960 0.940 0.976 0.940

=== Confusion Matrix ===

a b c <-- classified as 50 0 0 | a = Iris-setosa

0 47 3 | b = Iris-versicolor 0 3 47 | c = Iris-virginica

Figure 2.26: Output of *SMO* with a nonlinear kernel on the iris data, cont’d.

2.7. ATTRIBUTE SELECTION 67

=== Run information ===

Scheme: weka.classifiers.functions.Logistic -R 1.0E-8 -M -1 -num-decimal-places 4 Relation: iris Instances: 150 Attributes: 5sepallength

sepalwidth petallength petalwidth class Test mode: 10-fold cross-validation

=== Classifier model (full training set) ===

Logistic Regression with ridge parameter of 1.0E-8 Coefficients...

Class Variable Iris-setosa Iris-versicolor =============================================== sepallength 21.8065 2.4652 sepalwidth 4.5648 6.6809 petallength -26.3083 -9.4293 petalwidth -43.887 -18.2859 Intercept 8.1743 42.637

Odds Ratios...

Class Variable Iris-setosa Iris-versicolor =============================================== sepallength 2954196659.8892 11.7653 sepalwidth 96.0426 797.0304 petallength 0 0.0001 petalwidth 0 0

Time taken to build model: 0.01 seconds

=== Stratified cross-validation === === Summary ===

Correctly Classified Instances 144 96 % Incorrectly Classified Instances 6 4 % Kappa statistic 0.94 Mean absolute error 0.0287 Root mean squared error 0.1424 Relative absolute error 6.456 % Root relative squared error 30.2139 % Total Number of Instances 150

=== Detailed Accuracy By Class ===

TP Rate FP Rate Precision Recall F-Measure MCC ROC Area PRC Area Class 1.000 0.000 1.000 1.000 1.000 1.000 1.000 1.000 Iris-setosa 0.920 0.020 0.958 0.920 0.939 0.910 0.970 0.933 Iris-versicolor 0.960 0.040 0.923 0.960 0.941 0.911 0.975 0.933 Iris-virginica Weighted Avg. 0.960 0.020 0.960 0.960 0.960 0.940 0.982 0.955

=== Confusion Matrix ===

a b c <-- classified as 50 0 0 | a = Iris-setosa

0 46 4 | b = Iris-versicolor 0 2 48 | c = Iris-virginica

Figure 2.27: Output of *Logistic* for the iris data.

68 CHAPTER 2. THE EXPLORER

(b) The finished network with two hidden layers.

Figure 2.28: Using WEKA’s neural-network graphical user interface.

(a) Beginning the process of editing the network to add a second hidden layer.

2.7. ATTRIBUTE SELECTION 69

=== Run information ===

Scheme: weka.clusterers.SimpleKMeans -init 0 -max-candidates 100

-periodic-pruning 10000 -min-density 2.0 -t1 -1.25 -t2 -1.0 -N 2 -A ‘‘weka.core.EuclideanDistance -R first-last’’ -I 500 -num-slots 1 -S 10 Relation: weather Instances: 14 Attributes: 5outlook

temperature humidity windy play Test mode: evaluate on training data

=== Clustering model (full training set) ===

kMeans ======

Number of iterations: 3 Within cluster sum of squared errors: 16.237456311387238

Initial starting points (random):

Cluster 0: rainy,75,80,FALSE,yes Cluster 1: overcast,64,65,TRUE,yes

Missing values globally replaced with mean/mode

Final cluster centroids:

Cluster# Attribute Full Data 0 1 (14.0) (9.0) (5.0) ============================================== outlook sunny sunny overcast temperature 73.5714 75.8889 69.4 humidity 81.6429 84.1111 77.2 windy FALSE FALSE TRUE play yes yes yes

Time taken to build model (full training data) : 0 seconds

=== Model and evaluation on training set ===

Clustered Instances

0 9 ( 64%) 1 5 ( 36%)

Figure 2.29: Output of *SimpleKMeans* for the weather data.

70 CHAPTER 2. THE EXPLORER

=== Run information ===

Scheme: weka.clusterers.EM -I 100 -N 2 -X 10 -max -1

-ll-cv 1.0E-6 -ll-iter 1.0E-6 -M 1.0E-6 -K 10 -num-slots 1 -S 100 Relation: weather Instances: 14 Attributes: 5outlook

temperature humidity windy play Test mode: evaluate on training data

=== Clustering model (full training set) ===

EM==Number of clusters: 2 Number of iterations performed: 7

Cluster Attribute 0 1 (0.35) (0.65) ============================== outlook

sunny 3.8732 3.1268 overcast 1.7746 4.2254 rainy 2.1889 4.8111 [total] 7.8368 12.1632 temperature

mean 76.9173 71.8054 std. dev. 5.8302 5.8566

humidity

mean 90.1132 77.1719 std. dev. 3.8066 9.1962

windy

TRUE 3.14 4.86 FALSE 3.6967 6.3033 [total] 6.8368 11.1632 playyes 2.1227 8.8773 no 4.7141 2.2859 [total] 6.8368 11.1632

Time taken to build model (full training data) : 0 seconds

=== Model and evaluation on training set ===

Clustered Instances

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

Log likelihood: -9.13037Figure 2.30: Output of *EM* for the weather data.

2.7. ATTRIBUTE SELECTION 71

Figure 2.31: Attribute selection: specifying an evaluator and a search method.

72 CHAPTER 2. THE EXPLORER

**Chapter 3**

**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 on the panel shown in Figure 2.3a.

**3.1 Getting started**

Here is a step-by-step example that loads an ARFF file and performs a cross-validation using J48. We describe how to build up the final configuration shown in Figure 3.1. First create a source of data by expanding the *DataSources* folder in the *Design* palette and selecting *ARFFLoader*. 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 3.2a. Click *Configure* to get the dialog in Figure 3.2b, from which you can either browse for an ARFF file by clicking the *Browse* button, or type the path to one in the Filename field.

Now we specify which attribute is the class using a *ClassAssigner* object. This is found under the *Evaluation* folder in the *Design* palette, so expand the *Evaluation* folder, 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 3.2a. 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 in the Evaluation folder. Select it, place it on the canvas, and connect it to the class assigner by right-clicking the latter and selecting dataset from the menu (which is similar to that in Figure 3.2a). Next select J48 from the trees folder under the *Classifiers* folder and place a J48 component on the canvas. Connect J48 to the cross-validation fold maker in the usual way, but make the connection twice by first choosing *trainingSet* and then *testSet* from the pop-up menu for the cross-validation fold maker. The next step is to select a *ClassifierPerformanceEvaluator* from the *Evaluation*

73

74 CHAPTER 3. THE KNOWLEDGE FLOW INTERFACE

(b) File browser obtained from the *Configure* menu.

(a) Right-click menu.

Figure 3.2: Configuring a data source in the Knowledge Flow.

folder and connect J48 to it by selecting the *batchClassifier* entry from the pop-up menu for J48. Finally, from the *Visualization* folder we place a *TextViewer* component on the canvas. Connect

Figure 3.1: The Knowledge Flow interface.

3.2. KNOWLEDGE FLOW COMPONENTS 75

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 3.1 except that there is as yet no graph viewer. Start the flow of execution by clicking one of the two triangular-shaped “play” buttons at the left side of the main toolbar. The leftmost play button launches all data sources present in the flow in parallel; the other play button launches the data sources sequentially, where a particular order of execution can be specified by including a number at the start of the component’s name (a name can be set via the Set name entry on popup menu). For a small dataset things happen quickly. Progress information appears in the status area at the bottom of the interface. The entries in the status area show the progress of each step in the flow, along with their parameter settings (for learning schemes) and elapsed time. Any errors that occur in a processing step are shown in the status area by highlighting the corresponding row in red. 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 flow that you’ve just laboriously created is actually available (minus the *GraphViewer*) as a built-in *template*. Example templates can be accessed from the *Template* button, which is the third icon from the right in the toolbar at the top of the Knowledge Flow interface. There are a number of templates that come with WEKA, and certain packages, once installed via the package manager, add further ones to the menu. The majority of template flows can be executed without further modification as they have been configured to load datasets that come with the WEKA distribution.

**3.2 Knowledge Flow components**

Most of the Knowledge Flow components will be familiar from the Explorer. The *Classifiers* folder contains all of WEKA’s classifiers, the *Filters* folder contains the filters, the *Clusterers* folder holds the clusterers, the *AttSelection* folder contains evaluators and search methods for attribute selection, and the *Associations* panel holds the association rule learners. All components in the Knowledge Flow are run in a separate thread of execution, except in the case where data is being processed incrementally—in this case a single thread of execution is used because, generally, the amount of processing done per data point is small, and launching a separate thread to process each one would incur a significant overhead.

Possible data sources are ARFF files, XML ARFF files, JSON ARFF files, CSV files exported from spreadsheets, the C4.5 file format, databases, serialized instances, LibSVM and SVMLight data formats, and a special loader (*TextDirectoryLoader*) to load a directory of plain text files into a single set of instances. There is also a data source called *DataGrid* that allows the user to define attributes and enter the vaues of instances via a graphical interface. There is a data sink that corresponds to each data source, with the exception of the *TextDirectoryLoader* and *DataGrid*. In addition, there are data sinks that can save raw text, static image data and WEKA serialized models.

Under the *Visualization* folder, the *DataVisualizer* pops up a panel for visualizing data in a two-dimensional scatter plot as in Figure 2.6b, in which you can select the attributes you would

76 CHAPTER 3. THE KNOWLEDGE FLOW INTERFACE

like to see. *ScatterPlotMatrix* pops up a matrix of two-dimensional scatter plots for every pair of attributes, shown in Figure 2.16a. *AttributeSummarizer* gives a matrix of histograms, one for each attribute, like that in the lower right-hand corner of Figure 2.3b. *ModelPerformanceChart* draws ROC curves and other threshold curves. *CostBenefitAnalysis* allows interactive exploration of the tradeoffs in cost or benefit arising from different cost matrices. *GraphViewer*, used above, pops up a panel for visualizing tree-based models, as in Figure 2.6a. 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.

Under the *Evaluation* folder, *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.

Components under the *Flow* folder tend to affect the flow of data in some way. *FlowByExpres- sion* enables incoming data to be sent to two downstream components based on the evaluation of a logical expression. The expression can involve one or more clauses that test the value of an attribute in the incoming data against a constant or the value of another attribute. Multiple clauses can be and’ed or or’ed together. *Block*, when connected between two components, delays the passing of data to the downstream component until a user-specified component has finished processing. *InstanceStreamToBatchMaker* collects instances arriving in a stream from an incom- ing “instance” connection and produces a batch dataset when the last instance has arrived. This is particularly useful when placed after a reservoir sampling filter—it allows the instances output by reservoir sampling to be used to train batch learning schemes. The *Join* component performs an inner join on two incoming dataset or instance streams. The user specifies the key fields to join on for both datasets, and both are expected to arrive in sorted order of their key fields.

Under the *Tools* folder, *Sorter* sorts instances within batch datasets or instance streams according to the values of one or more user-specified attributes. It can handle datasets larger than can be fit into main memory by using instance connections and specifying an in-memory buffer size. The component implements a merge sort by writing the sorted in-memory buffer to a file when full, and then interleaving instances from the disk-based file(s) when the incoming instance stream has finished. *SubstringReplacer* replaces occurrences of substring or regular expression matches in incoming string attributes with user-specified values. *SubstringLabeler* matches substrings in a similar fashion, but instead of replacing them it creates a new attribute in the data that holds the value of a label specified by the user for each match condition.

3.3. CONFIGURING AND CONNECTING THE COMPONENTS 77

**3.3 Configuring and connecting the components**

You establish the knowledge flow by configuring the individual components and connecting them up. The menus that are available by right-clicking various component types have up to three sections: *Edit*, *Connections*, and *Actions*. The *Edit* operations delete components and open up their configuration panel. You can give a component a name by choosing *Set name* from the pop-up menu. Classifiers and filters are configured just as in the Explorer. Data sources are configured by opening a file (as we saw previously) or by setting a database connection, and evaluation components are configured by setting parameters such as the number of folds for cross-validation. 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 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* con- nections. 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 *Test- SetMaker* 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, provide 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* generates textual and graphical output for visu- alization 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*.

78 CHAPTER 3. THE KNOWLEDGE FLOW INTERFACE

**3.4 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: *SGD*, *SGDText*, *Hoeffd- ingTree*, *NaiveBayesMultinomialText*, a version of Naive Bayes (*NaiveBayesUpdateable*), instance- based learners (*IBk*, *KStar*, *LWL*), and *NaiveBayesMultinomialUpdateable*. All filters that work instance by instance are incremental: *Add*, *AddExpression*, *AddValues*, *ChangeDateFormat*, *Clas- sAssigner*, *Copy*, *FirstOrder*, *MakeIndicator*, *MergeTwoValues*, *MergeManyValues*, *NominalTo- Binary*, *NonSparseToSparse*, *NumericToBinary*, *NumericTransform*, *NumericCleaner*, *Obfus- cate*, *PartitionedMultiFilter*, *RandomSubset*, *Remove*, *RemoveByName*, *RemoveType*, *Remove- WithValues*, *RenameAttribute*, *RenameNominalValues*, *Reorder*, *ReplaceMissingWithUserCon- stant*, *ReservoirSample*, *SortLabels*, *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 3.3a shows a configuration that works incrementally. An instance connection is made from the loader to a class assigner component, which, in turn, is connected to the updatable Naive Bayes classifier. The classifier’s text output is taken to a viewer that gives a textual description 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 *StripChart* visualization component to generate a scrolling data plot.

Figure 3.3b shows the strip chart output. It plots accuracy, Kappa, 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.

3.4. INCREMENTAL LEARNING 79

(b) *StripChart* output.

Figure 3.3: A Knowledge Flow that operates incrementally.

(a) The configuration.

80 CHAPTER 3. THE KNOWLEDGE FLOW INTERFACE