Chapter 17

Introducing Machine Learning Concepts with WEKA

Tony C. Smith and Eibe Frank

Abstract

This chapter presents an introduction to data mining with machine learning. It gives an overview of various types of machine learning, along with some examples. It explains how to download, install, and run the WEKA data mining toolkit on a simple data set, then proceeds to explain how one might approach a bioinformatics problem. Finally, it includes a brief summary of machine learning algorithms for other types of data mining problems, and provides suggestions about where to find additional information.

Key words Machine learning, Data mining, WEKA, Bioinformatics, Tutorial

1 Data Mining and Machine Learning

A principal activity of all scientists is *trying to make sense of data*, and computers are often the primary means by which that is attempted. Spreadsheets and statistical packages are widely used tools for obtaining distributions, variances, and scatter plots, which can offer *some* insights; but there are other, much more difficult and important objectives of data analytics for which these metrics are of little utility. Biologists, for example, spend a good deal of time and effort looking for patterns in data that can be converted into an explanation of some phenomenon observed in nature. That is, the ultimate goal is to discover new, interesting, and potentially useful knowledge that furthers understanding of the natural world.

The process is one of first detecting regularities in data, then formulating some hypothesis as a characterization of those regularities, and finally testing the hypothesis against new data to see how robust it is. One might call such a process *data mining*—the search for valuable nuggets of insight in a slurry of observations. Done manually, this can be a tiring and frustratingly difficult thing to achieve; particularly when the amount of data involved is large. Fortunately, much of the process lends itself to automation, and computer scientists have devised a great many algorithms that can rapidly find and characterize patterns in data. Collectively, this type

Ewy Mathé and Sean Davis (eds.), *Statistical Genomics: Methods and Protocols*, Methods in Molecular Biology, vol. 1418, DOI 10.1007/978-1-4939-3578-9_17,

of computation is called *machine learning*, and it is increasingly a key tool for scientific discovery.

This chapter aims to provide the reader with a quick and practical introduction to data mining using machine learning software. It explains the different types of machine learning, provides an overview of some of the algorithms available to address these types of learning, shows how to download, install, and run one of the more widely used open-source data mining software tools—namely, WEKA [1]—and takes the reader through some examples demonstrating how to prepare data and load it into WEKA, how to run machine learning experiments on that data, and how to interpret the results. In short, by the end of this chapter the reader will be able to carry out machine learning.

Having said that, it is important to bear in mind that the depth of discussion here is still necessarily limited because the subject is very large. Like so many things, learning how to do data mining is not particularly difficult, but learning how to do it well is quite another thing. This chapter is thus a kind of *crash course* and a *small sampler*. Along the way, it attempts to alert the reader to some of the more common pitfalls that arise when carrying out machine learning experiments, and makes suggestions as to where more information can be obtained.

1.1 Structured vs. Unstructured Data

Before explaining how machine learning works, it is useful to get an idea of what its objectives are and what it is working with. There are many kinds of machine learning, and many different algorithms to choose from depending upon the data available and the goals of the study. We might start by first differentiating between *structured* data and *unstructured* data.

It is perhaps easiest to think of structured data as the kind of information one might record in a plain old spreadsheet. Each row of the spreadsheet is a single datum (i.e., an *instance*) and each column is a *feature* (i.e., or *attribute*) of the instances, such that each cell of the spreadsheet holds a value recorded for that feature for that instance—an *attribute-value* pair. The nature of the spreadsheet means that every instance is represented in the same structured way, even if some cells are empty or irrelevant or wrongly recorded. In contrast, unstructured data are things like documents (i.e., texts) or pictures (i.e., images) that have different dimensions, different formats, or other widely disparate characteristics.

Machine learning algorithms work well on structured data, and it is frequently the case that unstructured data simply has a structure imposed upon it prior to the actual learning. For example, a collection of images might be transformed into some regular set of attributes such that each image is generalized according to its colors, textures, shapes, intensities, dimensions, and so forth; and a collection of documents might see each text converted to something like a vector of counts for how often each word in the

complete lexicon of the entire collection occurs in each text. Unstructured learning is generally less common than structured learning, and has overhead associated with it that requires a fair bit of explanation, so its peculiarities are not covered in this chapter. We focus on learning from data as it may be found in a spreadsheet or a database table; but we include some examples where unstructured data (e.g., primary sequence data) is converted into tabular form.

1.2 Supervised vs. Unsupervised Learning

Another way to characterize data mining paradigms is to differentiate between *supervised* and *unsupervised* learning. In supervised learning, the data used for learning are treated as exemplars that have in some way been annotated with *the thing* to be learned. In such situations, the goal is to find a characterization of these examples so that judgments can later be made for new instances. In the simplest case, the value of one attribute of each example is the thing we want to predict (i.e., the judgment) and the assumption is that values of the other attributes allow for that if only the right relationship can be found. The learning algorithm seeks out a model of the data that supports such prediction. By learning on a subset of the existing data, we can test predictive accuracy on the remaining data; hence the learning is directed by examples provided by an expert and performance is supervised through evaluation against existing judgments. Whether or not performance on that test sample is indicative of future performance on novel data is an important question, and one we will address later.

1.3 Prediction: Classification and Regression

Supervised learning itself comes in two principal forms: classification and regression. If we are trying to predict something that has a discrete value, it is called classification. For example, we may want to predict whether the molecular composition of a blood sample (e.g., mass spectrum data) is or is not indicative of cancer; thus we have a binary classification problem. Or, we may have a sample of mitochondrial DNA and we want to predict the genus or species of the organism from which it most likely came; giving us a potentially very large number of classes to choose from in a so-called multiclass problem. And, if it is possible for an instance to belong to more than one class then we have what is known as a multilabel classification problem. For all classification problems, the objective is to accurately associate an instance with one or more labels from a finite set of alternatives. In comparison, if we are trying to predict a numeric value then the learning objective is to find some formula for generating a good estimate of the true value. In this situation, accuracy is measured by how close the estimate is to the actual value provided by the expert, rather than whether the predicted value is precisely right or not. This is regression, and typically requires different algorithms than those used for classification.

1.4 Clustering and Associations

In comparison, unsupervised learning is where we do not know the right answer ahead of time for any of the data—there is no prior basis to judge how good our result is. The goal is not to be able to make judgments that are right or wrong, per se, but simply to find interesting and useful generalities within the data.

A common form of unsupervised learning is that of *clustering*—given a collection of data, separate instances into two or more groups (clusters) based upon their similarities. Several issues arise in clustering that greatly affect the output and, consequently, dash any hope of an objective assessment about success. For one thing, it is generally not clear how many groups there should be. Should there be two? Twenty? One for each instance? Is it possible for one group be a subset of another (i.e., hierarchical), or overlap with other groups? Can an instance belong to more than one group at the same time and, if so, can membership be fractional or probabilistic?

In addition to this, a clustering algorithm must have criteria to decide whether two things are similar or dissimilar. Should they be in the same cluster or not? Similarity is not always as obvious as one would hope. Consider the situation where a child has been given a bunch of building blocks of all different shapes, sizes, and colors. One might find, after a time, that the child starts sorting them into groups; perhaps putting all the cubes together into one group, all the pyramids in another, all the cones in another, and so forth. Or maybe the child puts all the big blocks in one group, all the small ones in another, and all the rest in another. Or one might instead see the child sort all the blue ones together and all the red ones in another group, and so on, without regard to shape or size. Or the blocks may end up being sorted based upon which ones are the child's favorites, or which blocks are the newest, or which ones were played with most recently, or any other criterion that is difficult to assess objectively by a third party. Similarity is often pretty subjective, and the measure utilized greatly affects the outcome.

Another kind of unsupervised learning is one where the objective is simply to discover any interesting correlations within data. This is called association mining, and a classic example is that of so-called *basket analysis*, where supermarkets analyze the contents of shoppers' baskets at the checkout to see if the purchase of one item appears to correlate with the purchase of another; a discovery with the potential to help target future buyers through careful product placement. As an example from genomics, one might use association mining to look for correlations between genotypes and phenotypes, identify codependent genes, or find potentially interesting variant calls that co-occur.

Similar to clustering, the value of the output for this kind of unsupervised learning is difficult to assess and often specious or uninteresting. For example, one might be excited to discover that the presence of six variants correlates highly with both a specific

disease and the geographical area where someone lives, suggesting an inheritable genetic cause for that disease; but, one might be a little less excited to discover that the same data suggests a very high correlation between the town where someone lives and their nationality. Association mining algorithms can find correlations, but cannot judge their utility.

Clustering and association mining (i.e., unsupervised learning) are very different from classification and regression (i.e., supervised learning), and very different from each other, and specific algorithms have been devised to carry out these kinds of data mining tasks. Moreover, because classification and regression aim to predict something accurately, we can only use some of our data for learning and must withhold some for testing the result of that learning; and there are several important aspects to how one goes about partitioning the data for these two stages (i.e., training and testing). Unsupervised learning, on the other hand, has no objective measure of success, and therefore, all the data can be used as input to the algorithm.

Whatever kind of machine learning you are doing, it is important to remember that the algorithms are effectively heuristic. They rely on principled techniques (usually based upon statistical methods) to find patterns and express correlations, and they do this very well; but they cannot assess the true value of what they find. It is the scientist carrying out the machine learning study that must assess the quality and utility of the result. Put another way, machines do learning, but people do data mining.

1.5 Nominal and Numeric Data

One more important distinction to make relates to types of data. As mentioned earlier, some attributes have values that are discrete and chosen from a finite set. We can think of these as labels, or categories; but more generally we call them nominal. The color of a car, or the symbolic value of a nucleotide or amino acid in a primary sequence is an example. Nominal attributes typically only submit to tests of equality (e.g., "first amino acid in this peptide is Methionine, yes or no?", "base immediately upstream from this position is C or G, true or false?") and other comparisons like "greater than" or "has a value between X and Y" do not generally make sense. Values that are numbers can be tested for equality as well, but also submit to range tests and comparisons involving closeness (e.g., "how big is the difference between this value and some other one?"), while nominal values do not.

The *type* of a feature can place restrictions on which learning algorithms can be used, and what operations can be performed on that feature when formulating a generalization of it. For example, many algorithms construct a model of the data by considering how much knowing the value of a particular feature improves the probability of predicting the class of an instance. This is like the game of 20 questions, where the best question to ask next is the one that you

think improves your chance of guessing the right answer quickly. So it is that an algorithm will compute, for each attribute, the improvement in the probability of guessing the correct result (i.e., formally, the information gain) when this feature's value is known; then choose the best feature and use it as a test to partition the data into subsets; repeating the process for each subset until a sequence of attribute tests ends with a prediction.

For nominal attributes, only tests for equality to one or more of the observed values are possible. For numeric values, a learning algorithm must compute some constant against which an observed value can be compared (e.g., greater-than, less-than) to maximize information gain and partition the data into subsets. These are different processes, therefore WEKA needs to know whether it is dealing with a number or not. There are two other attribute types available in WEKA (dates and strings, which have their own peculiarities), but the vast majority of data types are typically nominal or numeric.

2 WEKA: Getting Started

Many machine learning systems exist, but the one we use in this chapter is the WEKA Machine Learning Workbench. It is one of the more popular open-source machine learning toolkits, and contains a wide range of learning algorithms. It is easily obtained simply by typing "download WEKA" into a search engine, which will likely bring up a link to the corresponding SourceForge project site, or to the University of Waikato (where WEKA was developed), which itself will take you to the SourceForge site. At the time of writing, the latest stable version is WEKA 3.6.12, but any version close to this will be sufficiently similar. You will want to select the version of WEKA that corresponds to your operating system (Windows x86 or 64 bit platforms, Mac or Linux).

WEKA is written entirely in Java, which means it can run on any computer that has a Java virtual machine (JVM) installed. If you are not sure whether your machine has a JVM, you can simply download the version of WEKA that includes Java VM 1.7 and when you go to install it your machine will tell you if you already have it and ask if you want to replace or update your JVM. A virtual machine is a piece of software that pretends to be a specific computer architecture that runs code specifically written for it (in this case, Java) so that such code can run on any machine with the corresponding virtual machine without having to recompile the code or fiddle with compatibility issues. It is what makes this kind of code very portable and easy to install and run.

Once downloaded, click on the downloaded file to start the install wizard that will guide you through the installation process. You will need about 65 MBytes of disk space, and the installation

Fig. 1 Start screen for WEKA

should be very quick. Once completed, if you have installed everything correctly and launched WEKA, you should get a startup window that looks like the one in Fig. 1. Do not worry if the size or layout is a little different, or if the typeface is not quite the same, as sometimes individual preferences on a machine will affect how the JVM displays things. The important thing is that you are looking at the same set of options.

For this tutorial, we will use the *Explorer*, which is the most straightforward way to use WEKA interactively. The *Experimenter* is a way to set up a series of machine learning experiments so that they can be saved and repeated in the future; *KnowledgeFlow* offers a drag-and-drop interface to set up a machine learning experiment by piecing together data sources, learning algorithms, training and test procedures and so forth with corresponding icons linked with arrows (which some people prefer); and the *Simple CLI* is a way to invoke parts of WEKA with type-written commands as one might do if setting up machine learning as part of a process pipe (e.g., where the input comes from a running process, or the output is required by a subsequent program). That is to say, WEKA is not just a stand-alone application, but has a complete API that allows it to be incorporated into other software systems using just those bits that are needed—either as a serial or concurrent thread, or by importing the classes of WEKA into another Java program. Unless you are a programmer yourself, this might not make a lot of sense, and is anyway beyond the scope of this chapter. To learn more about it, simply type "WEKA Simple CLI" into your favorite search engine and many useful links to explanations and tutorials are easily obtained.

2.1 Loading Data

If all has gone well, clicking on the Explorer option (so named because it is a way to go exploring with WEKA) will produce a new window similar to the one pictured in Fig. 2. Note the six tabs

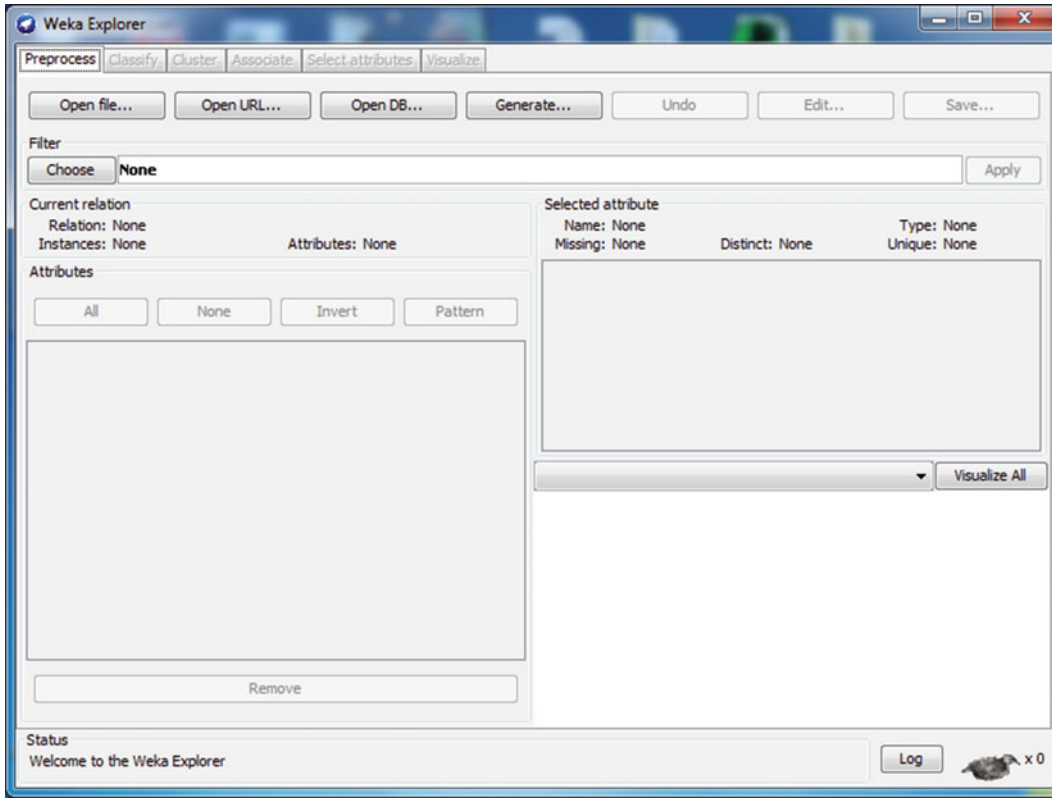

Fig. 2 Start screen for the WEKA Explorer

along the top, and you will see that you are in the Preprocess area. This is where interactive data mining begins. It is where you load data, examine it, perhaps modify it (more on that later) or otherwise get your data ready for input into a learning scheme. As suggested by the buttons along the top of the Preprocess window, data can be obtained from a file on your machine, or from a website via a URL, or it can be pulled out of a database if you are familiar with SQL, and you can even generate artificial data if you just want to experiment with the learning procedures more generally. We are only concerned here with loading data from a file, and the WEKA download actually includes a couple of dozen sample data files to experiment with.

Depending upon how search paths are set on your own machine, clicking on the *Open file* ... option should take you directly to the WEKA folder (i.e., directory) where the sample data, documentation and log folders are located; or you may have to traverse your directory structure to get there (for example, on a Windows machine, you would likely go to C:\Program Files (x86)\Weka-3-6 to find the data folder).

Use the "*Open file* ..." option to load the sample data file called "weather.numeric", and your WEKA window should end up

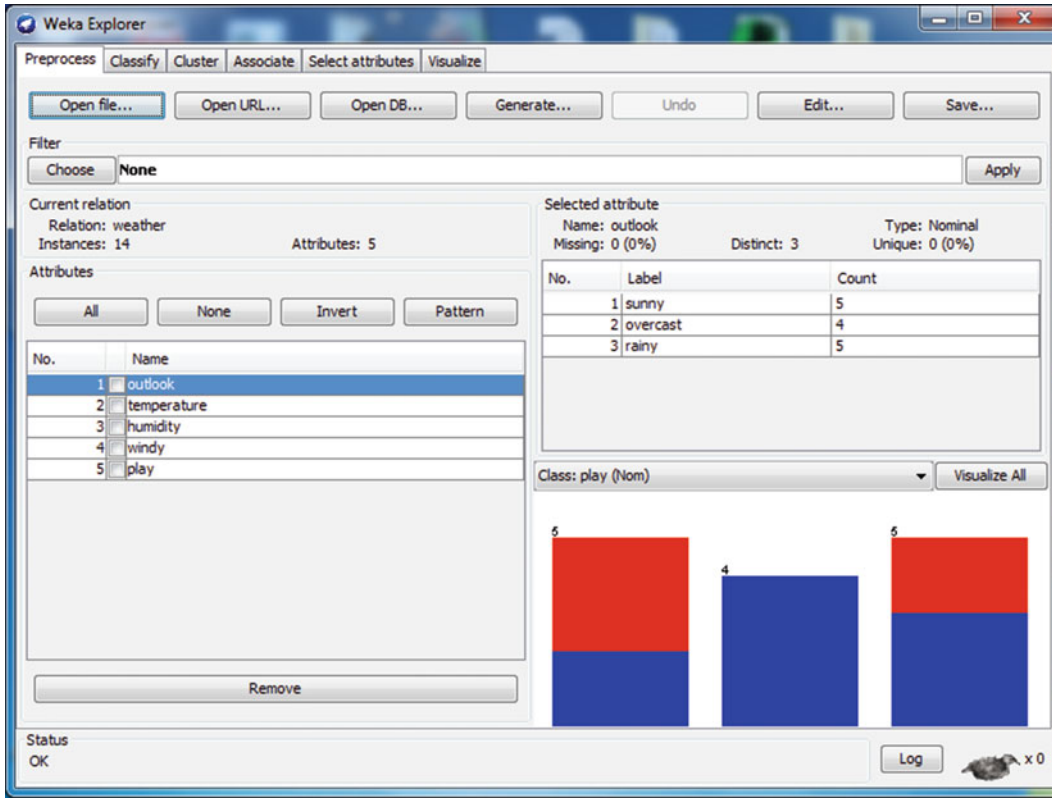

Fig. 3 After loading the weather.numeric sample data

looking like the one in Fig. 3. This data is contrived for demonstration purposes, supposedly recording weather conditions for 14 different days upon which some fictitious outdoor sport was either played or not played. The learning objective is to find a generalization of these past examples that helps decide if this game should be played on some future occasion given the prevailing conditions.

On the left side of the screen is information about the number of instances (i.e., examples) loaded from the file, how many attributes each instance has, and a checkbox list of those attributes. On the right hand side is some information about the specific attribute selected (i.e., highlighted) on the left, in this case the "outlook" attribute, and it includes information about its type (either nominal or numeric), how many distinct values for this attribute have been found in the data, what percentage of the instances are missing the value for this feature, and what percentage of the values only occur once (i.e., are unique). Below that is a table that lists the values observed for this feature and their frequencies (or, when a numeric attribute is selected, you will see the maximum and minimum values found for this attribute, and the mean and standard deviation), and below that is a colored bar chart showing the distribution for those values. Rolling the mouse over each bar brings up the

corresponding value (or, for numeric attributes, a range is shown). What are the colors in the bar chart? WEKA assumes that the user will likely be performing classification (although this is easily changed) and therefore one of the attributes on the left must be the target judgment; which WEKA further assumes is the last attributes (although this can be changed with the pulldown menu "Class:" just above the bar graph), which in this case is the nominal attribute called "play". If you select the "play" attribute on the left side of the panel (by clicking on the label) then you will see it only has two values: yes and no. These are given the colors blue and red, respectively (and these colors can be changed if you do not like the default scheme). Going back to the bar chart for *outlook* you can see that two of the bars are both red and blue while the third is all blue. This gives a quick perspective of how the target classes are distributed with respect to each of the attribute-value pairs. Right away we can see that if the outlook is overcast then the game has always been played in the past, but we may need to consider some other attribute when the outlook is sunny or rainy.

2.2 Preprocessing Data

Looking back to the left panel, under the "Attributes" label, you can see four buttons labeled "All", "None", "Invert", and "Pattern". These allow you to easily select subsets of attributes. There are many reasons why we might not want to keep all attributes for learning. Some may contain duplicate information of others (e.g., the *windy* attribute here is either true or false, so if we happened to also have a "*not windy*" attribute with the complement value then one of these two attributes could and should be tossed out as redundant), or some attributes may have too many missing values, or be too tightly related to the concept being learned to be of general use (e.g., if each instance had a unique identification code as an attribute, then knowing that code would let us perfectly predict the class simply by looking it up, but if a future instance has its own unique code then no generalization of this attribute would be useful for prediction). On the left side, you can use the checkboxes to select attributes, and the Remove button at the bottom allows you to delete them from consideration in subsequent learning. The buttons above the list of attributes are shortcuts to quickly select "All" attributes, "None" of the attributes, "Invert" your selection, or use a regular expression pattern to select a subset. These come in handy when a large number of attributes are being manipulated. For example, if you wanted to see what could be learned from just a few attributes in a data set that included hundreds of features, it is much easier to select the few you want to keep, invert your selection and then hit *Remove* than it is to select *one-by-one* the hundreds of attributes you do not want. Note that anything we do to the data on this panel does not affect the original data file, and we can repeal any operation with the *Undo* button (near the top). If we like the changes we've made and want

to keep them, we can always save them in a new file using the *Save* button, and we can also *Edit* individual values in the data, say, if we want to correct an error or fill in a missing value.

Near the top of the window, just below the top row of buttons, you can see a "Filter" section with a "Choose" button on one side and an "Apply" button on the other. This is where you can select from a very large number of filters that can transform data in a myriad of ways prior to learning. For example, you might want to take a nominal attribute whose value can be one of three possible labels and turn it into three new binary attributes that test each value individually instead. That is, if one attribute can have the value "A" or "B" or "C", we could turn it into three binary attributes called "Is A", "Is B", and "Is C" where one of these has the value "true" and the others "false" to reflect the original multi-valued attribute. This can occasionally be useful because it helps some learning algorithms search for characterizations more effectively.

Another common filter is to transform a numeric continuous valued attribute into a discrete valued one by binning values into a small number of categories defined by ranges. Many learning algorithms do this themselves, but do so heuristically and it may be the case that you know something about the data that would allow for better discretization, and this filter can be set up to do a better job of binning values prior to learning. In another situation, you might want to add a numeric feature that is the product or sum of two other numeric features (something that most learning algorithms do not do on their own because the possible combinations that could be tested are large), and this can be done with a filter. Indeed, just about anything you can think you might want to do, there is likely a filter for it in WEKA. Filtering can be extremely useful, but too complex to discuss in more detail in this chapter. Let us move on and get WEKA to learn something from the data we have loaded.

2.3 Choosing a Classifier

Predicting whether or not to play this game is a binary classification problem. Once we have prepared our data for learning, we select the *Classify* tab to move to the training and testing area. After selecting this tab you should see something similar to (but not exactly the same as) the screenshot shown in Fig. 4. There are two principal decisions to be made here that will most greatly affect the outcome of our machine learning experiment: what classifier to use, and what kind of testing to do for evaluation. The default classifier is ZeroR; also known as a *majority* classifier because it simply predicts the most frequent class found in the training data—which is often a good thing to check because if 90 % of your data belongs to one class and your machine learning experiment only gets 10 % wrong during testing, then you have not necessarily learned anything interesting because you are not doing any better than you would predicting the majority class.

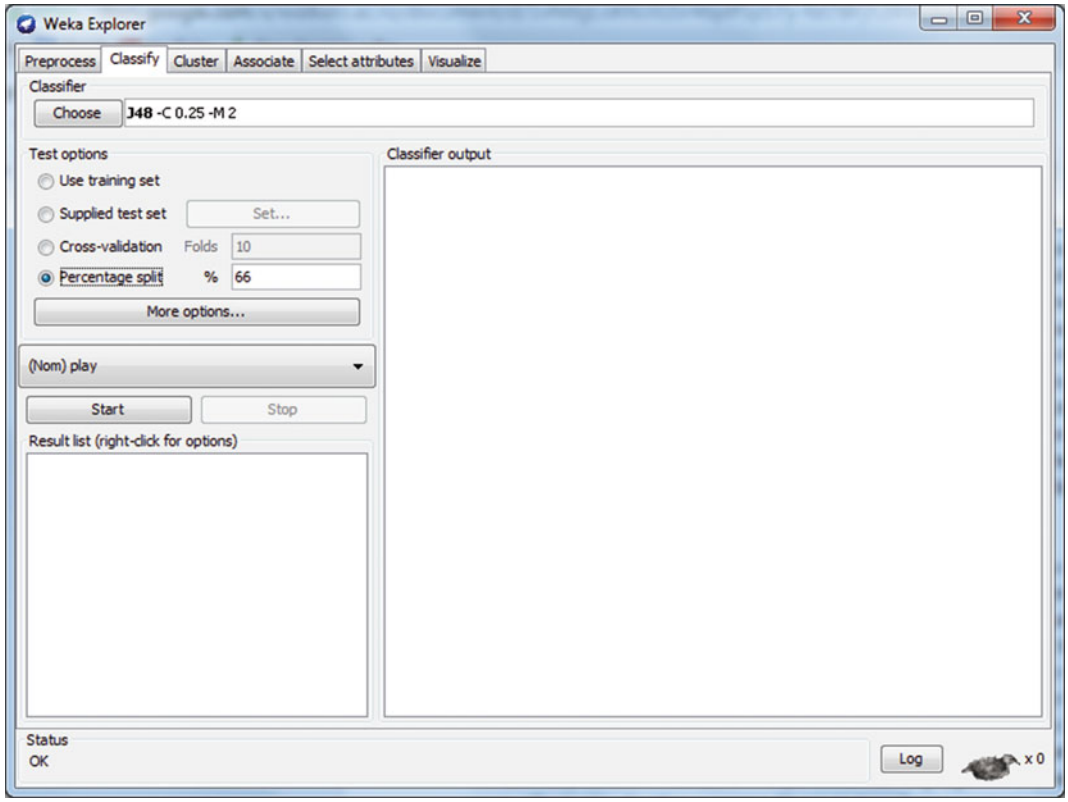

Fig. 4 Ready to learn with J48 and a 2:1 training-test split

For our first experiment, we want to use one of the most commonly used *decision tree* learners: J48 (an open-source version of Ross Quinlan's C4.5 algorithm [2]). Decision trees are easy to build and easy to use. They represent a simple top-down decision procedure, where one simply tests some condition first and, based on the result of that test, chooses another condition to test after that, and so on until you hit the end of a sequence of questions where a decision (i.e., judgment, or prediction) is finally made. It is a kind of *expert system*.

To choose the J48 classifier you click on the *Choose* button which brings up an expandable directory of classifiers. There are well over a hundred algorithms to choose from in WEKA, and they are organized according to the type of output they produce, the type of input they expect, and the type of decision process used to build the model. Some algorithms will be grayed out because they are not available for the kind of input you have prepared. For example, some algorithms cannot handle numeric input, and some produce a numeric prediction instead of a class label. J48 produces a tree, so it can be found under the *tree* subdirectory. Once selected it should appear in the classifier text box as shown in Fig. 4, along with a couple of default parameter settings (i.e., −C

0.25 −M 2, which set the confidence and minimum support required by the algorithm when building the tree). Almost every algorithm has parameters that can be altered to fine-tune or guide its behavior, and these can be accessed simply by clicking on the textbox where the current parameters are displayed. We have to stick with default parameters in this chapter, which are often satisfactory anyway.

2.4 Training and Testing

The next decision we have to make is how we are going to test the final model. You can see a list of *Test options* just underneath where you select the classifier, and the options are *Use training set*, *Supplied test set*, *Cross-validation*, and *Percentage split*.

Obviously, the learner needs examples to learn from. This is the training set. To assess performance of the final model, there needs to be some data that we can make predictions for and then compare those predictions against what is known for these data. This is the test set. If we use all of our data to train up a model, and then use the same data for testing, then we run the risk of over-estimating the value of the model because it has seen the test data during training. For example, if the model managed to memorize every training instance then it may likely get 100 % accuracy when testing on that same data. If the model has any generalization to it, then it may get a few wrong, but either way there are likely to be fewer surprises than if the test data had not been seen during learning. The result obtained by using the training data as test data gives rise to a thing called *resubstitution error*, and is often unjustifiably optimistic for predicting performance of the model on future novel data.

Another option is to have a completely separate test set stored in a separate file that is supplied only during testing. This is useful when the test data might be completely different than the training data in some fundamental way. For example, if one were trying to predict something about gram-negative bacteria based solely upon what is observed in gram-positive bacteria, the gram-positive training data could be loaded into WEKA at the outset to build the classifier, then the gram-negative data could be loaded from a separate file for testing; eliminating any risk the learner had a sneak peek.

Most often, we have one set of data and we must partition it into training and test sets. One way is to simply use a percentage split, using (say) 67 % of the data for training and test on the remaining 33 %. What percentage split to use is hard to generalize. The more data seen for training the better the chance of the learner finding a valid characterization for making predictions. On the other hand, the more data we have for testing, the more likely our success rate reflects the true virtue of the inferred model. Ideally we want an abundance of both training and test data, but in practice we just have whatever data we have.

A good way to make the most of our data is use all our data for training and all our data for testing, but not at the same time (as resubstitution error would ensue). To do this, we divide our data into a number of equal-sized subsets, called folds. For each fold, we remove it from the training set, build a model on the other folds and then test on the withheld portion. If we have k folds, then this is called k-fold cross-validation. That is, if k is 5 then we have fivefold cross-validation. We would withhold the first fold, train on the remaining four, then test on the first (which has not been seen by the learner). Next, we put the first fold back into the training set but take out the second fold and repeat training and testing. This is done once for each of the five folds, such that the test results are always for data not seen by the learner, and all the data is used for testing.

Cross-validation is widely regarded as the most reliable way to judge the results from machine learning when data are all in one set. How many folds should be used? Again, this is difficult to generalize. If too few folds are used then we may deprive the learner of enough data to build a good model. The maximum number of folds we can have would put one datum in each fold—so-called leave-one-out cross-validation (LOOCV)—and performance from this type of testing is on average closest to the true level of performance we can expect from the model on novel data. However, since each fold requires building a new model and evaluating with a new test set, too many folds with a large data set could take far too long to complete in a practical amount of time. A good compromise is to find a tractable value for k, then repeat k-fold cross-validation some number of times using different folds on each iteration, then average the results.

One final note about cross-validation: care should be taken to make sure that the distribution of classes in the test set is the same as in the training set. Imagine, for example, if 80 % of our data was in one class and 20 % in the other in a two-class problem. If we do fivefold cross-validation and all 20 % that comprise the minority class happen to end up in one fold then the learner would not have a chance to infer what differentiates the two classes. Keeping class distributions the same in training and test data is called stratification, and WEKA seeks to do this automatically. In a multi-class problem, however, if one class has very few exemplars then it may not be possible to maintain stratification if too many folds are used.

2.5 Running the Experiment

The weather data we loaded for this sample experiment has only 14 instances, so we set the test option to threefold cross-validation. Having chosen J48 (with default parameters) as our classifier, we can start the training and test phases in WEKA simply by pressing the START button in the middle of the left panel. For such a small data set, WEKA will complete these phases in the twinkling of an eye. For much larger and/or more complex data sets it could take a

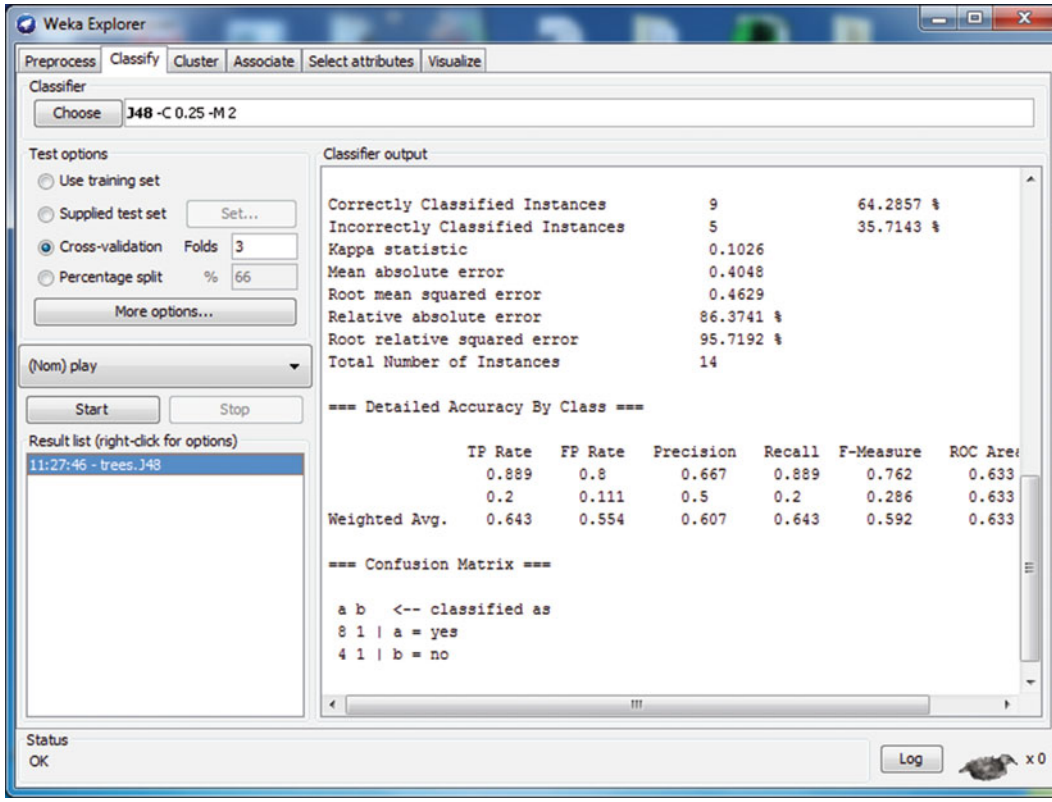

Fig. 5 Output after training and testing on weather data

lot longer. You can see if WEKA is still working because the icon of the bird in the bottom right corner (a picture of an actual Weka) will be in motion. If we notice progress going too slowly to complete in the time we have, we can simply interrupt it by hitting the STOP button, then rejig our parameters (or data) to try and make things tractable.

In this example experiment, WEKA completes in an instant and your display should look pretty much like the one shown in Fig. 5. The text window on the right side is now filled with details about the set-up of the experiment and the results from testing. At the top is a record of the classifier used and the parameter settings, some information about the data set, and the training and test procedure. Under that is a textual representation of the J48 tree. This is a bit obscure for people who are not computer scientists, but the basic method of interpretation is that each level of indentation (indicated by a vertical bar character) represents dropping a level in the tree. The decision criterion for the top level of the decision tree is flush to the left, and in this case it tests the "outlook" attribute. If the outlook is sunny, then evaluation proceeds down one level (indicated by one vertical bar) to test if humidity is greater or less than 75. If greater, then instead of dropping another level and testing

another attribute, the colon after the test shows the decision that is made at this point—in this case, "no" . . . do not play. The number (s) that appear in parentheses after the prediction reflect the number of instances that reached this point in the decision tree (and a second number here would state how many errors were made by this prediction). If the outlook is overcast then the decision of "yes" is made without any further tests (just as the distribution graph under the Preprocess tab showed), and if outlook is rainy then the windy attribute is also tested before a prediction is made.

This way of presenting a decision tree is pretty cryptic, but WEKA allows you to see a nice graphic version of the tree as well. On the bottom left of this window is the Result list. For each experiment you run, a new item will be added to this list. You can revisit the result of any experiment run during a session simply by clicking on the one you want to see. If you right-click on an experiment in this list, some other options appear for visualizing the results. To see an actual decision tree for this experiment, right-click on the experiment and select Visualize tree and a separate window will pop up with a picture similar to the one shown in Fig. 6.

For many (if not most) data sets for real world problems, the sheer size and complexity of the decision tree is too much to display in a window. WEKA will try anyway, and the result might be a very confused display as it tries to fit everything in. Right-clicking on the visualization window provides some re-display options that sometimes help.

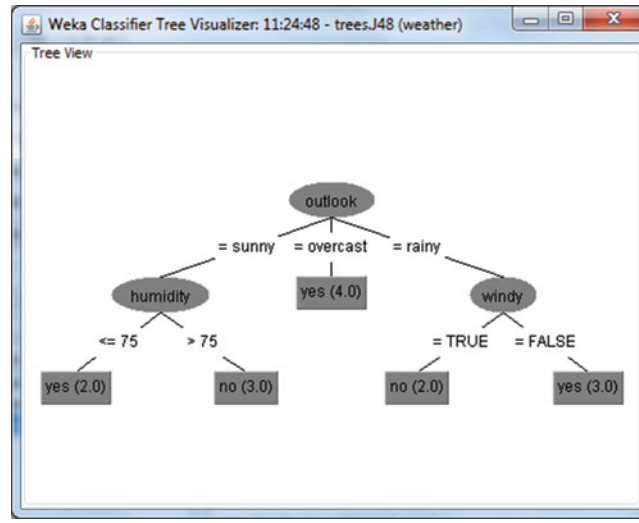

Fig. 6 The decision tree for the experiment from Fig. 5

2.6 Understanding the Results

Apart from a textual representation of the decision tree, the *Classifier output* area contains a number of performance statistics for the learning algorithm, all computed from the test sets used in the evaluation process. Note that the model that is shown—in this case, a decision tree—is built from the entire dataset loaded into the Preprocess panel. Thus, this model itself is not involved in the evaluation process at all (unless evaluation is performed on the training set or a separate test set). Rather, models built from subsets of the full data are used to establish values for the performance metrics. In a *k*-fold cross-validation, these metrics are aggregated across *k* test sets, which are used to evaluate *k* models built from the corresponding training (sub)sets.

Considering the values of the performance metrics, we can see that five instances in the data are misclassified in the cross-validation process. At the bottom of the output, in the *Summary* section, we can see how these errors are distributed across classes, in the so-called "confusion matrix". One instance has been incorrectly classified as class *no*, and four instances have been incorrectly classified as class *yes*. It is often useful to consider how the classifier compares to a random predictor that assigns instances randomly to classes (maintaining the same column totals in the confusion matrix). The kappa statistic, also shown in the *Summary* section, measures exactly this. A value of zero means that the classifier does not improve on the random straw man, which means that it has not learned anything useful from the data. One is the best possible value; it is achieved when all test instances are classified correctly.

WEKA also outputs error statistics, such as the squared error and the absolute error. These are the primary metrics for evaluating regression methods. In classification problems such as the example considered here, they measure the accuracy of the class probability estimates produced by the classifier. (Most algorithms in WEKA can produce a probability estimate for each class value when making a classification, and it can be useful to consider how accurate these probability estimates are.) WEKA also shows the relative absolute error and the root relative squared error. To compute the former, the mean absolute error of the classifier is divided by the mean absolute error obtained when simply predicting class probabilities based on the class frequencies in the training data, without considering any of the non-class attributes. The computation for the root relative squared error is performed analogously. These relative errors are expressed as percentages. Values close to 100 % mean that the classifier has not learned anything useful from the data.

The output area also shows a table of per-class performance statistics, with one row per class value. The corresponding target class for each row is called the "positive" class; all other classes are considered "negative". There is the *TP Rate* (true positive rate), which is the number of instances correctly assigned to the positive class, divided by the number of positive instances in the data; the *FP*

Rate (false positive rate), which is the number of negative instances incorrectly assigned to the positive class, divided by the number of negative instances; *Precision*, which is the number of instances correctly assigned to the positive class, divided by the total number of instances assigned to the positive class; and *Recall*, which is the same as *TP Rate*. The *F-Measure* is the harmonic mean of *Precision* and *Recall*. The last column contains the area under the ROC curve, an estimate of the probability that the classifier ranks a randomly chosen positive instances above a randomly chosen negative one, assuming its class probability estimates for the positive class are used for ranking. If the *ROC Area* is one, all positive instances are ranked above all negative ones—the ideal outcome—if it is zero, the classifier does not improve on randomly shuffling the data.

We have discussed the classification case here. When a regression method is evaluated, the same error statistics are output in the *Classifier output* area. Additionally, WEKA outputs Pearson's correlation coefficient, measuring the correlation between the predicted values and the actual values.

3 Approaching a Bioinformatics Problem

There are about as many bioinformatics problems as there are biologists, but a sizeable number start with primary sequence data. One class of problems has to do with characterizing some site in a set of sequences, such as glycosylation points [3] or inhibitor binding sites [4, 5]. For this example, we address the problem of identifying the cleavage point of the signal peptide.

Often, sequence data is obtained from a database in a FASTA format (or similar text-based form), such as the sample of three signal peptide cleavage records shown in Fig. 7. Each record has three lines where the first includes a description of the example sequence, the second is the primary sequence itself showing the complete signal peptide at the beginning of the nascent protein along with the first 30 residues of the mature protein, and the third line is a character-mask mapping one-to-one with the preceding primary sequence using the letter S to indicate the corresponding residue above is part of the signal peptide, M indicates a residue in the mature protein and C marks the cleavage site as the first residue of the mature protein. The problem is stated in this way: given a new sequence, identify the cleavage site. But, before we can begin mining this data, we must transform it into a format amenable as input to a learning algorithm, such as the spreadsheet form discussed earlier, where each row is an instance and each column is an attribute, one of which is the attribute to be predicted. We assume only the sequence itself will be available when predicting the cleavage point in a future novel instance.

```
  50 11S3_HELAN     20 11S GLOBULIN SEED STORAGE PROTEIN G3 PRECURSOR
MASKATLLLAFTLLFATCIARHQQRQQQQNQCQLQNIEALEPIEVIQAEA
SSSSSSSSSSSSSSSSSSSCMMMMMMMMMMMMMMMMMMMMMMMMMMM
  51 11SB_CUCMA     21 11S GLOBULIN BETA SUBUNIT PRECURSOR.
MARSSLFTFLCLAVFINGCLSQIEQQSPWEFQGSEVWQQHRYQSPRACRLE
SSSSSSSSSSSSSSSSSSSSCMMMMMMMMMMMMMMMMMMMMMMMMMMM
  54 1B39_HUMAN     24 HLA CLASS I HISTOCOMPATIBILITY ANTIGEN, BW-42 B*4201 ALP
MLVMAPRTVLLLLSAALALTETWAGSHSMRYFYTSVSRPGRGEPRFISVGYVDD
SSSSSSSSSSSSSSSSSSSSSSCMMMMMMMMMMMMMMMMMMMMMMMMMM
...
```

Fig. 7 Signal peptide cleavage data

If we assume there are some physicochemical properties of the nascent protein that determine the cleavage point, then our goal is to use machine learning to find a characterization of those properties from known examples. Well, really our goal might be simply to predict the cleavage site of a new protein accurately; but, we might instead be *more* interested in the characterization itself (i.e., the model) as it may lead to understanding and new knowledge about the cleavage process. These are two different aspirations and typically suggest we might use different learning algorithms based on what we want at the end. For output with explanatory power, we might want a decision tree like the one in Fig. 6 as they are easy to understand. If we are more interested in predictive accuracy, a support vector machine or random forest is typically a better choice because they do not need to make a decision based on a linear sequence of tests; however, the models they produce are often difficult to interpret and thus not useful as "knowledge".

The next choice we have to make before we prepare the data for learning is how to frame the learning objective as an attribute of each instance. In the case of this cleavage problem, we could try to predict the length of the peptide, and therefore each instance is one complete sequence. This requires a learning algorithm that is capable of predicting a number from properties embodied in an entire sequence. One reason we might not take this approach is that, as mentioned earlier, numeric prediction algorithms tend to seek an estimate with a small error margin and do not often produce exactly the right value. Their performance is evaluated in terms of that margin of error. We need to be right or wrong, not close enough, here.

Another way to view the problem—one that is probably more useful—is to predict whether or not a given residue is the first of the mature protein, which is a binary classification problem. In this case, each instance is one residue at a given point in the sequence.

Having framed the objective, we now must generate a set of features to describe each instance. If we pursue the binary classification approach then we want to characterize each residue with attributes that we think are relevant to whether or not it marks the cleavage point. If we have no idea what properties might be relevant then we may start naively using whatever we can think of. For example, if we suspect that the residues on either side of the cleavage point play a critical role, then we might simply create (say) six features that record the residue at each of the three positions upstream and downstream from the cleavage site. As it happens, distributional analysis of residue frequency in known peptide data shows alanine and serine are commonly found in the −1 and −3 positions (i.e., one and three residues upstream of the cleavage site), and a couple of other residues are also unusually frequent at these positions, so some traction might be gained from these features. However, given 20 possible residues at each of these six positions, there are 20^6 combinations of values that could be seen, times two for the class attribute (i.e., YES or NO, this is a cleavage site), or 128 million possible instances in a comprehensive data set. In practice, one would be lucky to have a couple of thousand instances to learn from, presenting potential for a *data sparseness* problem, where so many combinations are not seen by the learner that it is almost trivial to find some characterization that is pretty accurate simply because there are no instances to disprove it. This is not to say that such a characterization is necessarily wrong. If the data we possess happens to contain the necessary information for inferring a correct generalization of the underlying principle, it is said to be a "characteristic set". The problem is that when data sparseness exists it is virtually impossible to tell whether an accurate model is correct or just lucky.

Biologists typically know *something* about the data that has a reasonable chance of factoring into a good generalization, and this is a good basis for formulating a set of features. For example, it is known that signal peptides are pretty short, having an average length of about 23 residues. Thus it might be useful to have a feature that records how far the residue in question is from the start of the nascent protein; but, this too might yield too many numeric values to generalize easily. Given that the vast majority of signal peptide lengths are within six or seven of the mean, it may be more fruitful to have a binary feature that simply records if the residue in question lies within this range; or, one might record how many standard deviations from the mean it is, thus reducing the range of possible values and mitigating possible sparseness.

Rather than using specific amino acid labels as feature values, one could reduce the combinatoric explosion by recording more general physicochemical properties for them. This is also helpful because the hypothesis we seek from the data mining process is one that generalizes according to such properties anyway. Moreover,

existing research about signal peptides has identified a number of regularities based on such properties. For example, the region of about six residues upstream of the cleavage site is generally not very hydrophobic, while the region of about eight residues upstream of that is typically quite hydrophobic. Thus we might have two binary features, one for each of these regions, whose value is "true" if the region is hydrophobic or "false" if it is not. Or each could be a numeric attribute quantifying just how hydrophobic a region is.

Once a set of features has been devised to describe each instance in our dataset, there is the challenge of actually producing them. Functions and macros in spreadsheets can often do the job, or any programming language will suffice; but obviously some modicum of computer programming skill is required. Sometimes the best solution for a biologist is just to make friends with a good computer scientist.

One more issue needs to be addressed, however, before we can proceed to computing the final data set. Successful learning needs not only positive instances of the concept to be learned, but negative ones as well, in order to eliminate overly general models like "predict every residue marks the cleavage site", which will be 100 % accurate on our data. If negative instances are available, then great. But many biological studies end up with just samples of the thing demonstrating the characteristic of interest.

In the signal peptide problem, negative instances are simply every other residue in our sequences that does not mark the start of a mature protein. Given the lengths of the sample sequences, this gives us about 75 times as many negative instances as positive ones. If we give them all to the learning algorithm there is a good chance that it will have a difficult time inferring a model that does better than simply predicting the majority class (i.e., predict that every residue is NOT at the cleavage site and the prediction is wrong only 1 in 75 times). In a binary classification problem, it is generally good to have the same number of positive and negative instances.

In this specific case, we might want to create negative instances by choosing residues *near* the cleavage point so that the properties are similar to positive instances, forcing the learner to find a discriminating model. It is probably also a good idea to generate several samples of negative data and run the machine learning experiment with each to avoid putting too much stock in an outcome that only works well serendipitously for the particular sample of negative instances we selected.

Once all features have been generated for all the data, it can be saved in a comma-separated-values (CSV) file and this can be loaded directly into WEKA. When WEKA imports a CSV file, it assumes the first line of input is a list of feature names/labels (which is typically the case for a CSV file saved from a spreadsheet) that it uses to name each attribute in the "Preprocess" window and in the resulting output after learning. The rest of the lines are taken to be

instances. Recall, however, that it is important to differentiate between nominal and numeric features. When reading a CSV file, WEKA attempts to parse every value as numeric, and if this succeeds for every value associated with a feature then that feature is assumed to be numeric, otherwise it is assumed to be nominal. There are many situations where this assumption can fail, such as when one feature-value that is supposed to be numeric has a stray character that prevents it from being parsed as a number, then the whole set of numbers associated with that feature are treated as nominal labels. Another situation is when a numeric value is not really being used as a number and is better treated as being nominal, as with an identification code. To solve the first problem, you're best to go back to the procedure that generated the data and fix the error. For the second type of problem, WEKA needs to be informed that what looks like a numeric attribute is actually a nominal one. WEKA's native file format for data is the AARF format, which is very similar to a CSV file but has some header information specifying types and ranges of data (and some other things). If loading a CSV file leads WEKA to misinterpret data types then a quick way to fix this is to save the data (using the "Save" button on the "Preprocess" tab) which generates an *ARFF* version that you can then edit directly to change type information.

4 Other Kinds of Learning

WEKA provides a uniform user interface to a large number of machine learning algorithms. This is useful because it is often difficult to determine a priori which algorithm is most suitable for a particular dataset and will produce the highest accuracy or most pertinent patterns. Decision tree learning is attractive because trees can be grown very quickly, even for large datasets, and may provide valuable insights. A closely related approach is to learn classification rules [6]: sets of simple if-then rules that describe the conditions (i.e., attribute-value combinations) under which a certain classification is to be made. Rule sets can be learned almost as quickly as decision trees but may provide a more compact representation of the salient relationships in the data. In WEKA, algorithms of this type can be found in the "rules" package, just as tree learners can be found in the "trees" package.

Trees and rules are not the only models that are intelligible and, thus, potentially able to provide useful insights. Another option is to learn a graphical model of the probability distribution underlying the data. A Bayesian network (or belief network) is a type of graphical model in the form of a directed graph [7]. Each node in this graph corresponds to an attribute in the data and directed edges connecting these nodes encode statistical dependencies between the attributes. Cycles are not allowed in this graph.

If there is a directed edge from node A to node B, node A is called a "parent" of node B. Each node has a conditional probability distribution attached to it that gives the probability of each of the node's attribute values given those of its parent nodes. Learning a Bayesian network amounts to determining suitable parent nodes for each node and estimating a conditional probability distribution based on the chosen parents. With nominal data, estimating the latter is as simple as counting how often a particular attribute value occurs for each combination of values of the parent nodes. The trickier part is to determine appropriate connections between nodes; various search methods are employed for this. The goal is to find an accurate model with as few edges as possible because experience has shown that simpler networks are more likely to reflect causal relationships between attributes in the data.

Bayesian networks can be used for supervised learning as well as for unsupervised learning. WEKA employs them for the former task, where they are used to estimate probabilities for the class attribute. A particularly simple form of Bayesian network is the naive Bayes model [8], which has a predetermined structure and only requires estimation of conditional probabilities. In this model, each attribute, excluding the class attribute, has exactly one parent, namely the class attribute. This corresponds to the assumption that the values of the other attributes are statistically independent given a particular value for the class attribute (i.e., considering the data for a particular class, knowing the value for one attribute does not give us any information regarding the values of other attributes). Learning a naive Bayes model is thus extremely fast. Moreover, in spite of its simplicity, it often yields accurate classifications, making it a very popular choice amongst practitioners. General Bayesian networks, as well as naive Bayes, are available in the "bayes" package for WEKA.

The above techniques can provide valuable insight, but in some applications the goal is to simply maximize predictive accuracy on new data. In this case, learning algorithms from WEKA's "functions" package are worth investigating. This package provides algorithms for learning basic linear models, such as linear regression and logistic regression, but also contains several more sophisticated methods. Two very prominent models implemented in this package, which often yield high accuracy, are support vector machines [9] and multilayer perceptrons [10]. Both are models that can be expressed as standard mathematical functions. Given an input instance, they produce an output by applying a sequence of mathematical operations. In the regression case, this output can be directly used as the prediction; in the classification case, it needs to be compared to a threshold to determine what the classification should be. A multilayer perceptron is a type of artificial neural network, where the component functions—the nodes of the network—mimic the behavior of neurons in the brain. The number

of nodes in the network, their arrangement into layers, and the connections between them, are predefined by the user. Learning the network amounts to determining appropriate parameter values for the component functions. This is done by applying a gradient-based optimization technique to minimize the error on the training data, starting with randomly chosen parameter values.

A conceptual drawback of multilayer perceptrons is that the final model found using gradient-based optimization depends on the initial parameter values because the error function contains local minima. Support vector machines are theoretically appealing because their optimization process converges to a unique solution. Just like multilayer perceptrons, they can be used to represent nonlinear relationships in the data. This is possible because the model can be expressed in terms of dot products between instances. The training instances that form part of the solution expressed in this way are called the "support vectors". Support vector machines are able to model nonlinear relationships by replacing the simple dot product with a so-called "kernel function." A kernel function implicitly maps the instances into a higher-dimensional space and then takes the dot product in this space, yielding nonlinear behavior in the original instance space. It is best to think of kernel functions as similarity functions with this particular property. Different kernel functions exist and can be plugged in when configuring the learning algorithm in WEKA. A popular kernel function that often works well is the Gaussian radial basis function kernel.

The idea of using similarity functions, or distance functions, for machine learning has been around much longer than support vector machines. One of the oldest machine learning methods is the nearest neighbor classifier [11]. The basic version simply stores the training data; to make a prediction for a test instance, it simply finds the most similar training instance and predicts its class value. This can be made more robust by combining class values from the k most similar instances, yielding the k nearest neighbor classifier. Because the basic version of this method simply stores the training data and only does serious computation when predictions are to be made, it is called a "lazy" learning method. Implementations of this method, and closely related methods, are to be found in WEKA's "lazy" package. WEKA supports various distance functions to measure similarity and also provides advanced data structures such as KD trees [12] to speed up the search for the nearest neighbor. A drawback of all lazy learning methods is that they do not generate a model that provides insight into the data.

The biggest variety of supervised learning algorithms is located in WEKA's "meta" package. It contains algorithms that wrap around user-specified base learners, which are themselves supervised learning algorithms. For example, it contains algorithms for bagging [13], boosting [14], and randomization [15] that all generate a so-called "ensemble" classifier by repeatedly invoking a

base learner, e.g., a decision tree learner. The resulting ensemble classifier is often significantly more accurate than the base learner by itself. However, any kind of interpretability will be lost.

Apart from ensemble classifiers, the meta package also provides several useful wrappers for tasks such as performing cost-sensitive learning, combining multiple different learning algorithms by a simple voting process or a more powerful "stacking" approach [16]—where the process of combining their predictions is phrased as another learning problem—and performing attribute selection prior to learning. Cost-sensitive learning is useful in practice when different types of classification errors incur different costs. This can be accommodated by adapting the model itself [17], or its predictions [18]. Attribute selection [19] is useful when the data contains a large number of attributes and only a small subset is likely to be of predictive value.

The vast majority of algorithms in WEKA perform supervised learning (i.e., classification or regression), but the workbench also contains algorithms for clustering and association rule mining (performed interactively in WEKA by selecting the appropriate tab at the top of the window). Classical algorithms for clustering are *k*-means [20], which yields a flat set of clusters represented by cluster centroids, and hierarchical clustering [21], which yields a dendrogram of clusters that can be displayed graphically. A probabilistic alternative to k-means is to fit a Gaussian mixture model to the data [22], where each Gaussian represents a cluster. The seminal Apriori algorithm for association rule mining [23] is also available in WEKA, with several other, more recent, algorithms for mining such rules efficiently by applying sophisticated data structures to speed up the search for frequent patterns in the data.

5 Concluding Remarks

Biological research yields tremendous amounts of data; machine learning provides tools that can help to make sense of this data and turn it into actionable models. The content of this chapter provides a brief, high-level introduction to basic concepts and algorithms in machine learning, using the WEKA workbench to illustrate how these can be applied. WEKA makes it relatively easy to enter the world of machine learning because it provides graphical user interfaces that do not require any scripting or serious programming. Readers who do not mind getting their hands dirty by writing scripts may also want to take a look at machine learning libraries for the statistical computing environment R [24] or the scikit-learn library for the programming language Python [25]. Just like WEKA, they come with extensive documentation.

References

1. Witten IH, Frank E, Hall MA (2011) Data mining: practical machine learning tools and techniques, 3rd edn. Morgan Kaufmann, Burlington, MA
2. Ross Quinlan J (1993) C 4.5: programs for machine learning. Morgan Kaufmann, San Mateo, CA
3. Blom N, Sicheritz-Pontén T, Gupta R, Gammeltoft S, Brunak S (2004) Prediction of post-translational glycosylation and phosphorylation of proteins from the amino acid sequence. Proteomics 4:1633–1649
4. Ramana J, Gupta D (2010) Machine learning methods for prediction of CDK-inhibitors. PLoS One 5(10):e13357
5. Buchwald F, Richter L, Kramer S (2011) Predicting a small molecule- kinase interaction map: a machine learning approach. J Cheminform 3:22
6. Fürnkranz J (1999) Separate-and-conquer rule learning. Artif Intell Rev 13(1):3–54
7. Friedman N, Geiger D, Goldszmidt M (1997) Bayesian network classifiers. Mach Learn 29(2-3):131–163
8. Domingos P, Pazzani M (1997) On the optimality of the simple Bayesian classifier under zero-one loss. Mach Learn 29(2–3):103–130
9. Cortes C, Vapnik V (1995) Support-vector networks. Mach Learn 20(3):273–297
10. Rumelhart DE, Hinton GE, Williams RJ (1986) Learning representations by back-propagating errors. Nature 323(9):533–536
11. Cover TM, Hart PE (1967) Nearest neighbor pattern classification. IEEE Trans Inform Theory 13(1):21–27
12. Friedman JH, Bentley JL, Finkel RA (1977) An algorithm for finding best matches in logarithmic expected time. ACM Trans Math Softw 3 (3):209–226
13. Breiman L (1996) Bagging predictors. Mach Learn 24(2):123–140
14. Freund Y, Schapire RE (1996) Experiments with a new boosting algorithm. International conference on machine learning. Morgan Kaufmann, Bari, Italy
15. Dietterich TG (2000) Ensemble methods in machine learning. Multiple classifier systems. Springer, Berlin, pp 1–15
16. Wolpert DH (1992) Stacked generalization. Neural Netw 5(2):241–259
17. Ting KM (1998) Inducing cost-sensitive trees via instance weighting. Principles of data mining and knowledge discovery. Springer, Berlin, pp 139–147
18. Duda RO, Hart PE (1973) Pattern classification and scene analysis, vol 3. Wiley, New York
19. Kohavi R, John GH (1997) Wrappers for feature subset selection. Artif Intell 97 (1):273–324
20. Hartigan JA (1975) Clustering algorithms. Wiley, New York
21. Johnson SC (1967) Hierarchical clustering schemes. Psychometrika 32(3):241–254
22. McLachlan GJ, Basford KE (1987) Mixture models: inference and applications to clustering. CRC, New York
23. Rakesh A, Srikant R (1994) Fast algorithms for mining association rules. International conference on very large databases. Morgan Kaufmann, Santiago de Chile, Chile
24. Ihaka R, Gentleman R (1996) R: a language for data analysis and graphics. J Comput Graph Stat 5(3):299–314
25. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011) Scikit-learn: machine learning in Python. J Mach Learn Res 12:2825–2830