Machine learning is a burgeoning new technology for mining knowledge from data, a technology that a lot of people are beginning to take seriously. We don’t want to oversell it. The kind of machine learning we know is not about the big problems: futuristic visions of autonomous robot servants, philosophical conundrums of consciousness, metaphysical issues of free will, evolutionary—or theological—questions of where intelligence comes from, linguistic debates over language learning, psychological theories of child development, or cognitive explanations of what intelligence is and how it works. For us, it’s far more prosaic: machine learning is about algorithms for inferring structure from data and ways of validating that structure. These algorithms are not abstruse and complicated, but they’re not completely obvious and trivial either.

Looking forward, the main challenge ahead is applications. Opportunities abound. Wherever there is data, things can be learned from it. Whenever there is too much data for people to pore over themselves, the mechanics of learning will have to be automatic. But the inspiration will certainly not be automatic! Applications will come not from computer programs, nor from machine
learning experts, nor from the data itself, but from the people who work with
the data and the problems from which it arises. That is why we have written
this book, and the Weka system described in Part II—to empower those who
are not machine learning experts to apply these techniques to the problems that
arise in daily working life. The ideas are simple. The algorithms are here. The
rest is really up to you!

Of course, development of the technology is certainly not finished. Machine
learning is a hot research topic, and new ideas and techniques continually
emerge. To give a flavor of the scope and variety of research fronts, we close Part
I by looking at some topical areas in the world of data mining.

8.1 Learning from massive datasets

The enormous proliferation of very large databases in today’s companies and
scientific institutions makes it necessary for machine learning algorithms to
operate on massive datasets. Two separate dimensions become critical when any
algorithm is applied to very large datasets: space and time.

Suppose the data is so large that it cannot be held in main memory. This
causes no difficulty if the learning scheme works in an incremental fashion,
processing one instance at a time when generating the model. An instance can
be read from the input file, the model can be updated, the next instance can be
read, and so on—without ever holding more than one training instance in main
memory. Normally, the resulting model is small compared with the dataset size,
and the amount of available memory does not impose any serious constraint
on it. The Naïve Bayes method is an excellent example of this kind of algorithm;
there are also incremental versions of decision tree inducers and rule learning
schemes. However, incremental algorithms for some of the learning methods
described in this book have not yet been developed. Other methods, such as
basic instance-based schemes and locally weighted regression, need access to all
the training instances at prediction time. In that case, sophisticated caching and
indexing mechanisms have to be employed to keep only the most frequently
used parts of a dataset in memory and to provide rapid access to relevant
instances in the file.

The other critical dimension when applying learning algorithms to massive
datasets is time. If the learning time does not scale linearly (or almost linearly)
with the number of training instances, it will eventually become infeasible to
process very large datasets. In some applications the number of attributes is a
critical factor, and only methods that scale linearly in the number of attributes
are acceptable. Alternatively, prediction time might be the crucial issue. Fortu-
nately, there are many learning algorithms that scale gracefully during both
training and testing. For example, the training time for Naïve Bayes is linear in
both the number of instances and the number of attributes. For top-down decision tree inducers, we saw in Section 6.1 (pages 196–198) that training time is linear in the number of attributes and, if the tree is uniformly bushy, log-linear in the number of instances (if subtree raising is not used or, if it is, with a further log factor).

When a dataset is too large for a particular learning algorithm to be applied, there are three ways to make learning feasible. The first is trivial: instead of applying the scheme to the full dataset, use just a small subset for training. Of course, information is lost when subsampling is employed. However, the loss may be negligible because the predictive performance of a learned model often flattens out long before all the training data is incorporated into it. If this is the case, it can easily be verified by observing the model’s performance on a holdout test set for training sets of different size.

This kind of behavior, called the law of diminishing returns, may arise because the learning problem is a simple one, so that a small volume of training data is sufficient to learn an accurate model. Alternatively, the learning algorithm might be incapable of grasping the detailed structure of the underlying domain. This is often observed when Naïve Bayes is employed in a complex domain: additional training data may not improve the performance of the model, whereas a decision tree’s accuracy may continue to climb. In this case, of course, if predictive performance is the main objective you should switch to the more complex learning algorithm. But beware of overfitting! Take care not to assess performance on the training data.

Parallelization is another way of reducing the time complexity of learning. The idea is to split the problem into smaller parts, solve each using a separate processor, and combine the results together. To do this, a parallelized version of the learning algorithm must be created. Some algorithms lend themselves naturally to parallelization. Nearest-neighbor methods, for example, can easily be distributed among several processors by splitting the data into several parts and letting each processor find the nearest neighbor in its part of the training set. Decision tree learners can be parallelized by letting each processor build a subtree of the complete tree. Bagging and stacking (although not boosting) are naturally parallel algorithms. However, parallelization is only a partial remedy because with a fixed number of processors, the algorithm’s asymptotic time complexity cannot be improved.

A simple way to apply any algorithm to a large dataset is to split the data into chunks of limited size and learn models separately for each one, combining the result using voting or averaging. Either a parallel bagging-like scheme or a sequential boosting-like scheme can be employed for this purpose. Boosting has the advantage that new chunks can be weighted based on the classifiers learned from previous chunks, thus transferring knowledge between chunks. In both cases memory consumption increases linearly with dataset size; hence some
form of pruning is necessary for very large datasets. This can be done by setting aside some validation data and only adding a model from a new chunk to the committee classifier if it increases the committee’s performance on the validation set. The validation set can also be used to identify an appropriate chunk size by running the method with several different chunk sizes in parallel and monitoring performance on the validation set.

The best but most challenging way to enable a learning paradigm to deal with very large datasets would be to develop new algorithms with lower computational complexity. In some cases, it is provably impossible to derive exact algorithms with lower complexity. Decision tree learners that deal with numeric attributes fall into this category. Their asymptotic time complexity is dominated by the sorting process for the numeric attribute values, a procedure that must be performed at least once for any given dataset. However, stochastic algorithms can sometimes be derived that approximate the true solution but require a much smaller amount of time.

Background knowledge can make it possible to vastly reduce the amount of data that needs to be processed by a learning algorithm. Depending on which attribute is the class, most of the attributes in a huge dataset might turn out to be irrelevant when background knowledge is taken into account. As usual, it pays to carefully engineer the data that is passed to the learning scheme and make the greatest possible use of any prior information about the learning problem at hand. If insufficient background knowledge is available, the attribute filtering algorithms described in Section 7.1 can often drastically reduce the amount of data—possibly at the expense of a minor loss in predictive performance. Some of these—for example, attribute selection using decision trees or the 1R learning scheme—are linear in the number of attributes.

Just to give you a feeling for the amount of data that can be handled by straightforward implementations of machine learning algorithms on ordinary microcomputers, we ran the decision tree learner J4.8 on a dataset with 600,000 instances, 54 attributes (10 numeric and 44 binary), and a class with seven values. We used a Pentium 4 processor with a 2.8-GHz clock and a Java virtual machine with a “just-in-time compiler.” It took 40 minutes to load the data file, build the tree using reduced-error pruning, and classify all the training instances. The tree had 20,000 nodes. Note that this implementation is written in Java, and executing a Java program is often several times slower than running a corresponding program written in C because the Java byte-code must be translated into machine code before it can be executed. (In our experience the difference is usually a factor of three to five if the virtual machine uses a just-in-time compiler.)

There are datasets today that truly deserve the adjective massive. Scientific datasets from astrophysics, nuclear physics, earth science, and molecular biology are measured in hundreds of gigabytes—or even terabytes. So are datasets
containing records of financial transactions. Application of standard programs for machine learning to such datasets in their entirety is a very challenging proposition.

8.2 Incorporating domain knowledge

Throughout this book we have emphasized the importance of getting to know your data when undertaking practical data mining. Knowledge of the domain is absolutely essential for success. Data about data is often called metadata, and one of the frontiers in machine learning is the development of schemes to allow learning methods to take metadata into account in a useful way.

You don’t have to look far for examples of how metadata might be applied. In Chapter 2 we divided attributes into nominal and numeric. But we also noted that many finer distinctions are possible. If an attribute is numeric an ordering is implied, but sometimes there is a zero point and sometimes not (for time intervals there is, but for dates there is not). Even the ordering may be nonstandard: angular degrees have an ordering different from that of integers because 360° is the same as 0° and 180° is the same as −180° or indeed 900°. Discretization schemes assume ordinary linear ordering, as do learning schemes that accommodate numeric attributes, but it would be a routine matter to extend them to circular orderings. Categorical data may also be ordered. Imagine how much more difficult our lives would be if there were no conventional ordering for letters of the alphabet. (Looking up a listing in the Hong Kong telephone directory presents an interesting and nontrivial problem!) And the rhythms of everyday life are reflected in circular orderings: days of the week, months of the year. To further complicate matters there are many other kinds of ordering, such as partial orderings on subsets: subset A may include subset B, subset B may include subset A, or neither may include the other. Extending ordinary learning schemes to take account of this kind of information in a satisfactory and general way is an open research problem.

Metadata often involves relations among attributes. Three kinds of relations can be distinguished: semantic, causal, and functional. A semantic relation between two attributes indicates that if the first is included in a rule, the second should be, too. In this case, it is known a priori that the attributes only make sense together. For example, in agricultural data that we have analyzed, an attribute called milk production measures how much milk an individual cow produces, and the purpose of our investigation meant that this attribute had a semantic relationship with three other attributes, cow-identifier, herd-identifier, and farmer-identifier. In other words, a milk production value can only be understood in the context of the cow that produced the milk, and the cow is further linked to a specific herd owned by a given farmer. Semantic relations
are, of course, problem dependent: they depend not just on the dataset but also on what you are trying to do with it.

*Causal* relations occur when one attribute causes another. In a system that is trying to predict an attribute caused by another, we know that the other attribute must be included to make the prediction meaningful. For example, in the agricultural data mentioned previously there is a chain from the farmer, herd, and cow identifiers, through measured attributes such as milk production, down to the attribute that records whether a particular cow was retained or sold by the farmer. Learned rules should recognize this chain of dependence.

*Functional* dependencies occur in many databases, and the people who create databases strive to identify them for the purpose of normalizing the relations in the database. When learning from the data, the significance of a functional dependency of one attribute on another is that if the latter is used in a rule there is no need to consider the former. Learning schemes often rediscover functional dependencies that are already known. Not only does this generate meaningless, or more accurately tautological, rules, but also other, more interesting patterns may be obscured by the functional relationships. However, there has been much work in automatic database design on the problem of inferring functional dependencies from example queries, and the methods developed should prove useful in weeding out tautological rules generated by learning schemes.

Taking these kinds of metadata, or prior domain knowledge, into account when doing induction using any of the algorithms we have met does not seem to present any deep or difficult technical challenges. The only real problem—and it is a big one—is how to express the metadata in a general and easily understandable way so that it can be generated by a person and used by the algorithm.

It seems attractive to couch the metadata knowledge in just the same representation as the machine learning scheme generates. We focus on rules, which are the norm for much of this work. The rules that specify metadata correspond to prior knowledge of the domain. Given training examples, additional rules can be derived by one of the rule induction schemes we have already met. In this way, the system might be able to combine “experience” (from examples) with “theory” (from domain knowledge). It would be capable of confirming and modifying its programmed-in knowledge based on empirical evidence. Loosely put, the user tells the system what he or she knows, gives it some examples, and it figures the rest out for itself!

To make use of prior knowledge expressed as rules in a sufficiently flexible way, it is necessary for the system to be able to perform logical deduction. Otherwise, the knowledge has to be expressed in precisely the right form for the learning algorithm to take advantage of it, which is likely to be too demanding for practical use. Consider causal metadata: if A causes B and B causes C, then we would like the system to deduce that A causes C rather than having to state that fact explicitly. Although in this simple example explicitly stating the new
fact presents little problem, in practice, with extensive metadata, it will be unrealistic to expect the system’s users to express all logical consequences of their prior knowledge.

A combination of deduction from prespecified domain knowledge and induction from training examples seems like a flexible way of accommodating metadata. At one extreme, when examples are scarce (or nonexistent), deduction is the prime (or only) means of generating new rules. At the other, when examples are abundant but metadata is scarce (or nonexistent), the standard machine learning techniques described in this book suffice. Practical situations span the territory between.

This is a compelling vision, and methods of inductive logic programming, mentioned in Section 3.6, offer a general way of specifying domain knowledge explicitly through statements in a formal logic language. However, current logic programming solutions suffer serious shortcomings in real-world environments. They tend to be brittle and to lack robustness, and they may be so computation intensive as to be completely infeasible on datasets of any practical size. Perhaps this stems from the fact that they use first-order logic, that is, they allow variables to be introduced into the rules. The machine learning schemes we have seen, whose input and output are represented in terms of attributes and constant values, perform their machinations in propositional logic without variables—greatly reducing the search space and avoiding all sorts of difficult problems of circularity and termination. Some aspire to realize the vision without the accompanying brittleness and computational infeasibility of full logic programming solutions by adopting simplified reasoning systems. Others place their faith in the general mechanism of Bayesian networks, introduced in Section 6.7, in which causal constraints can be expressed in the initial network structure and hidden variables can be postulated and evaluated automatically. It will be interesting to see whether systems that allow flexible specification of different types of domain knowledge will become widely deployed.

8.3 Text and Web mining

Data mining is about looking for patterns in data. Likewise, text mining is about looking for patterns in text: it is the process of analyzing text to extract information that is useful for particular purposes. Compared with the kind of data we have been talking about in this book, text is unstructured, amorphous, and difficult to deal with. Nevertheless, in modern Western culture, text is the most common vehicle for the formal exchange of information. The motivation for trying to extract information from it is compelling—even if success is only partial.

The superficial similarity between text and data mining conceals real differences. In Chapter 1 we characterized data mining as the extraction of implicit,
previously unknown, and potentially useful information from data. With text mining, however, the information to be extracted is clearly and explicitly stated in the text. It is not hidden at all—most authors go to great pains to make sure that they express themselves clearly and unambiguously. From a human point of view, the only sense in which it is “previously unknown” is that time restrictions make it infeasible for people to read the text themselves. The problem, of course, is that the information is not couched in a manner that is amenable to automatic processing. Text mining strives to bring it out in a form suitable for consumption by computers or by people who do not have time to read the full text.

A requirement common to both data and text mining is that the information extracted should be potentially useful. In one sense, this means actionable—capable of providing a basis for actions to be taken automatically. In the case of data mining, this notion can be expressed in a relatively domain-independent way: actionable patterns are ones that allow nontrivial predictions to be made on new data from the same source. Performance can be measured by counting successes and failures, statistical techniques can be applied to compare different data mining methods on the same problem, and so on. However, in many text mining situations it is hard to characterize what “actionable” means in a way that is independent of the particular domain at hand. This makes it difficult to find fair and objective measures of success.

As we have emphasized throughout this book, “potentially useful” is often given another interpretation in practical data mining: the key for success is that the information extracted must be comprehensible in that it helps to explain the data. This is necessary whenever the result is intended for human consumption rather than (or as well as) for automatic action. This criterion is less applicable to text mining because, unlike data mining, the input itself is comprehensible. Text mining with comprehensible output is tantamount to summarizing salient features from a large body of text, which is a subfield in its own right: text summarization.

We have already encountered one important text mining problem: document classification, in which each instance represents a document and the instance’s class is the document’s topic. Documents are characterized by the words that appear in them. The presence or absence of each word can be treated as a Boolean attribute, or documents can be treated as bags of words, rather than sets, by taking word frequencies into account. We encountered this distinction in Section 4.2, where we learned how to extend Naïve Bayes to the bag-of-words representation, yielding the multinomial version of the algorithm.

There is, of course, an immense number of different words, and most of them are not very useful for document classification. This presents a classic feature selection problem. Some words—for example, function words, often called stopwords—can usually be eliminated a priori, but although these occur very
frequently there are not all that many of them. Other words occur so rarely that they are unlikely to be useful for classification. Paradoxically, infrequent words are common—nearly half the words in a typical document or corpus of documents occur just once. Nevertheless, such an overwhelming number of words remains after these word classes are removed that further feature selection may be necessary using the methods described in Section 7.1. Another issue is that the bag- (or set-) of-words model neglects word order and contextual effects. There is a strong case for detecting common phrases and treating them as single units.

Document classification is supervised learning: the categories are known beforehand and given in advance for each training document. The unsupervised version of the problem is called document clustering. Here there is no predefined class, but groups of cognate documents are sought. Document clustering can assist information retrieval by creating links between similar documents, which in turn allows related documents to be retrieved once one of the documents has been deemed relevant to a query.

There are many applications of document classification. A relatively easy categorization task, language identification, provides an important piece of metadata for documents in international collections. A simple representation that works well for language identification is to characterize each document by a profile that consists of the n-grams, or sequences of n consecutive letters, that appear in it. The most frequent 300 or so n-grams are highly correlated with the language. A more challenging application is authorship ascription in which a document’s author is uncertain and must be guessed from the text. Here, the stopwords, not the content words, are the giveaway, because their distribution is author dependent but topic independent. A third problem is the assignment of key phrases to documents from a controlled vocabulary of possible phrases, given a large number of training documents that are tagged from this vocabulary.

Another general class of text mining problems is metadata extraction. Metadata was mentioned previously as data about data: in the realm of text the term generally refers to salient features of a work, such as its author, title, subject classification, subject headings, and keywords. Metadata is a kind of highly structured (and therefore actionable) document summary. The idea of metadata is often expanded to encompass words or phrases that stand for objects or “entities” in the world, leading to the notion of entity extraction. Ordinary documents are full of such terms: phone numbers, fax numbers, street addresses, email addresses, email signatures, abstracts, tables of contents, lists of references, tables, figures, captions, meeting announcements, Web addresses, and more. In addition, there are countless domain-specific entities, such as international standard book numbers (ISBNs), stock symbols, chemical structures, and mathematical equations. These terms act as single vocabulary items, and many document processing tasks can be significantly improved if they are identified
as such. They can aid searching, interlinking, and cross-referencing between documents.

How can textual entities be identified? Rote learning, that is, dictionary lookup, is one idea, particularly when coupled with existing resources—lists of personal names and organizations, information about locations from gazetteers, or abbreviation and acronym dictionaries. Another is to use capitalization and punctuation patterns for names and acronyms; titles (Ms.), suffixes (Jr.), and baronial prefixes (von); or unusual language statistics for foreign names. Regular expressions suffice for artificial constructs such as uniform resource locators (URLs); explicit grammars can be written to recognize dates and sums of money. Even the simplest task opens up opportunities for learning to cope with the huge variation that real-life documents present. As just one example, what could be simpler than looking up a name in a table? But the name of the Libyan leader Muammar Qaddafi is represented in 47 different ways on documents that have been received by the Library of Congress!

Many short documents describe a particular kind of object or event, combining entities into a higher-level composite that represent the document’s entire content. The task of identifying the composite structure, which can often be represented as a template with slots that are filled by individual pieces of structured information, is called information extraction. Once the entities have been found, the text is parsed to determine relationships among them. Typical extraction problems require finding the predicate structure of a small set of predetermined propositions. These are usually simple enough to be captured by shallow parsing techniques such as small finite-state grammars, although matters may be complicated by ambiguous pronoun references and attached prepositional phrases and other modifiers. Machine learning has been applied to information extraction by seeking rules that extract fillers for slots in the template. These rules may be couched in pattern-action form, the patterns expressing constraints on the slot-filler and words in its local context. These constraints may involve the words themselves, their part-of-speech tags, and their semantic classes.

Taking information extraction a step further, the extracted information can be used in a subsequent step to learn rules—not rules about how to extract information but rules that characterize the content of the text itself. These rules might predict the values for certain slot-fillers from the rest of the text. In certain tightly constrained situations, such as Internet job postings for computing-related jobs, information extraction based on a few manually constructed training examples can compete with an entire manually constructed database in terms of the quality of the rules inferred.

The World Wide Web is a massive repository of text. Almost all of it differs from ordinary “plain” text because it contains explicit structural markup. Some
markup is internal and indicates document structure or format; other markup
is external and defines explicit hypertext links between documents. These informa-
tion sources give additional leverage for mining Web documents. Web mining
is like text mining but takes advantage of this extra information and often
improves results by capitalizing on the existence of topic directories and other
information on the Web.

Internet resources that contain relational data—telephone directories or
product catalogs—use hypertext markup language (HTML) formatting com-
mands to clearly present the information they contain to Web users. However,
it is quite difficult to extract data from such resources automatically. To do so,
existing software systems use simple parsing modules called wrappers to analyze
the page structure and extract the requisite information. If wrappers are coded
by hand, which they often are, this is a trivial kind of text mining because it
relies on the pages having a fixed, predetermined structure from which infor-
mation can be extracted algorithmically. But pages rarely obey the rules. Their
structures vary; Web sites evolve. Errors that are insignificant to human readers
throw automatic extraction procedures completely awry. When change occurs,
adjusting a wrapper manually can be a nightmare that involves getting your
head around the existing code and patching it up in a way that does not cause
breakage elsewhere.

Enter wrapper induction—learning wrappers automatically from examples.
The input is a training set of pages along with tuples representing the informa-
tion derived from each page. The output is a set of rules that extracts the tuples
by parsing the page. For example, it might look for certain HTML delimiters—
paragraph boundaries (<p>), list entries (<li>), or boldface (<b>)—that the Web
page designer has used to set off key items of information, and learn the
sequence in which entities are presented. This could be accomplished by iterat-
ing over all choices of delimiters, stopping when a consistent wrapper is encoun-
tered. Then recognition will depend only on a minimal set of cues, providing
some defense against extraneous text and markers in the input. Alternatively,
one might follow Epicurus’s advice at the end of Section 5.9 and seek a robust
wrapper that uses multiple cues to guard against accidental variation. The great
advantage of automatic wrapper induction is that when errors are caused by
stylistic variants it is simple to add these to the training data and reinduce a new
wrapper that takes them into account. Wrapper induction reduces recognition
problems when small changes occur and makes it far easier to produce new sets
of extraction rules when structures change radically.

A development called the semantic Web aims to enable people to publish
information in a way that makes its structure and semantics explicit so that
it can be repurposed instead of merely read. This would render wrapper
induction superfluous. But if and when the semantic Web is deployed, the
requirement for manual markup—not to mention the huge volumes of legacy pages—will likely increase the demand for automatic induction of information structure.

Text mining, including Web mining, is a burgeoning technology that is still, because of its newness and intrinsic difficulty, in a fluid state—akin, perhaps, to the state of machine learning in the mid-1980s. There is no real consensus about what it covers: broadly interpreted, all natural language processing comes under the ambit of text mining. It is usually difficult to provide general and meaningful evaluations because the mining task is highly sensitive to the particular text under consideration. Automatic text mining techniques have a long way to go before they rival the ability of people, even without any special domain knowledge, to glean information from large document collections. But they will go a long way, because the demand is immense.

8.4 Adversarial situations

A prime application of machine learning is junk email filtering. As we write these words (in late 2004), the scourge of unwanted email is a burning issue—maybe by the time you read them the beast will have been vanquished or at least tamed. At first blush junk email filtering appears to present a standard problem of document classification: divide documents into “ham” and “spam” on the basis of the text they contain, guided by training data, of which there are copious amounts. But it is not a standard problem because it involves an adversarial aspect. The documents that are being classified are not chosen randomly from an unimaginably huge set of all possible documents; they contain emails that are carefully crafted to evade the filtering process, designed specifically to beat the system.

Early spam filters simply discarded messages containing “spammy” words that connote such things as sex, lucre, and quackery. Of course, much legitimate correspondence concerns gender, money, and medicine: a balance must be struck. So filter designers recruited Bayesian text classification schemes that learned to strike an appropriate balance during the training process. Spammers quickly adjusted with techniques that concealed the spammy words by misspelling them; overwhelmed them with legitimate text, perhaps printed in white on a white background so that only the filter saw it; or simply put the spam text elsewhere, in an image or a URL that most email readers download automatically.

The problem is complicated by the fact that it is hard to compare spam detection algorithms objectively; although training data abounds, privacy issues preclude publishing large public corpora of representative email. And there are strong temporal effects. Spam changes character rapidly, invalidating sensitive
statistical tests such as cross-validation. Finally, the bad guys can also use machine learning. For example, if they could get hold of examples of what your filter blocks and what it lets through, they could use this as training data to learn how to evade it.

There are, unfortunately, many other examples of adversarial learning situations in our world today. Closely related to junk email is search engine spam: sites that attempt to deceive Internet search engines into placing them prominently in lists of search results. Highly ranked pages yield direct financial benefits to their owners because they present opportunities for advertising, providing strong motivation for profit seekers. Then there are the computer virus wars, in which designers of viruses and virus-protection software react to one another’s innovations. Here the motivation tends to be general disruption and denial of service rather than monetary gain.

Computer network security is a continually escalating battle. Protectors harden networks, operating systems, and applications, and attackers find vulnerabilities in all three areas. Intrusion detection systems sniff out unusual patterns of activity that might be caused by a hacker’s reconnaissance activity. Attackers realize this and try to obfuscate their trails, perhaps by working indirectly or by spreading their activities over a long time—or, conversely, by striking very quickly. Data mining is being applied to this problem in an attempt to discover semantic connections among attacker traces in computer network data that intrusion detection systems miss. This is a large-scale problem: audit logs used to monitor computer network security can amount to gigabytes a day even in medium-sized organizations.

Many automated threat detection systems are based on matching current data to known attack types. The U.S. Federal Aviation Administration developed the Computer Assisted Passenger Pre-Screening System (CAPPS), which screens airline passengers on the basis of their flight records and flags individuals for additional checked baggage screening. Although the exact details are unpublished, CAPPS is, for example, thought to assign higher threat scores to cash payments. However, this approach can only spot known or anticipated threats. Researchers are using unsupervised approaches such as anomaly and outlier detection in an attempt to detect suspicious activity. As well as flagging potential threats, anomaly detection systems can be applied to the detection of illegal activities such as financial fraud and money laundering.

Data mining is being used today to sift through huge volumes of data in the name of homeland defense. Heterogeneous information such as financial transactions, health-care records, and network traffic is being mined to create profiles, construct social network models, and detect terrorist communications. This activity raises serious privacy concerns and has resulted in the development of privacy-preserving data mining techniques. These algorithms try to discern patterns in the data without accessing the original data directly,
typically by distorting it with random values. To preserve privacy, they must guarantee that the mining process does not receive enough information to reconstruct the original data. This is easier said than done.

On a lighter note, not all adversarial data mining is aimed at combating nefarious activity. Multiagent systems in complex, noisy real-time domains involve autonomous agents that must both collaborate in a team and compete against antagonists. If you are having trouble visualizing this, think soccer. Robo-soccer is a rich and popular domain for exploring how machine learning can be applied to such difficult problems. Players must not only hone low-level skills but must also learn to work together and adapt to the behavior patterns of different opponents.

Finally, machine learning has been used to solve a historical literary mystery by unmasking a prolific author who had attempted to conceal his identity. As Koppel and Schler (2004) relate, Ben Ish Chai was the leading rabbinic scholar in Baghdad in the late nineteenth century. Among his vast literary legacy are two separate collections of about 500 Hebrew-Aramaic letters written in response to legal queries. He is known to have written one collection. Although he claims to have found the other in an archive, historians suspect that he wrote it, too, but attempted to disguise his authorship by deliberately altering his style. The problem this case presents to machine learning is that there is no corpus of work to ascribe to the mystery author. There were a few known candidates, but the letters could equally well have been written by anyone else. A new technique appropriately called unmasking was developed that creates a model to distinguish the known author’s work A from the unknown author’s work X, iteratively removes those features that are most useful for distinguishing the two, and examines the speed with which cross-validation accuracy degrades as more features are removed. The hypothesis is that if work X is written by work A’s author, who is trying to conceal his identity, whatever differences there are between work X and work A will be reflected in only a relatively small number of features compared with the differences between work X and the works of a different author, say the author of work B. In other words, when work X is compared with works A and B, the accuracy curve as features are removed will decline much faster for work A than it does for work B. Koppel and Schler concluded that Ben Ish Chai did indeed write the mystery letters, and their technique is a striking example of the original and creative use of machine learning in an adversarial situation.

8.5 Ubiquitous data mining

We began this book by pointing out that we are overwhelmed with data. Nowhere does this affect the lives of ordinary people more than on the World Wide Web. At present, the Web contains more than 5 billion documents, total-
ing perhaps 20 TB—and it continues to grow exponentially, doubling every 6 months or so. Most U.S. consumers use the Web. None of them can keep pace with the information explosion. Whereas data mining originated in the corporate world because that’s where the databases are, text mining is moving machine learning technology out of the companies and into the home. Whenever we are overwhelmed by data on the Web, text mining promises tools to tame it. Applications are legion. Finding friends and contacting them, maintaining financial portfolios, shopping for bargains in an electronic world, using data detectors of any kind—all of these could be accomplished automatically without explicit programming. Already text mining techniques are being used to predict what link you’re going to click next, to organize documents for you, and to sort your mail. In a world where information is overwhelming, disorganized, and anarchic, text mining may be the solution we so desperately need.

Many believe that the Web is but the harbinger of an even greater paradigm shift: ubiquitous computing. Small portable devices are everywhere—mobile phones, personal digital assistants, personal stereo and video players, digital cameras, mobile Web access. Already some devices integrate all these functions. They know our location in physical time and space, help us communicate in social space, organize our personal planning space, recall our past, and envelop us in global information space. It is easy to find dozens of processors in a middle-class home in the U.S. today. They do not communicate with one another or with the global information infrastructure—yet. But they will, and when they do the potential for data mining will soar.

Take consumer music. Popular music leads the vanguard of technological advance. Sony’s original Walkman paved the way to today’s ubiquitous portable electronics. Apple’s iPod pioneered large-scale portable storage. Napster’s network technology spurred the development of peer-to-peer protocols. Recommender systems such as Firefly brought computing to social networks. In the near future content-aware music services will migrate to portable devices. Applications for data mining in networked communities of music service users will be legion: discovering musical trends, tracking preferences and tastes, and analyzing listening behaviors.

Ubiquitous computing will weave digital space closely into real-world activities. To many, extrapolating their own computer experiences of extreme frustration, arcane technology, perceived personal inadequacy, and machine failure, this sounds like a nightmare. But proponents point out that it can’t be like that, because, if it is, it won’t work. Today’s visionaries foresee a world of “calm” computing in which hidden machines silently conspire behind the scenes to make our lives richer and easier. They’ll reach beyond the big problems of corporate finance and school homework to the little annoyances such as where are the car keys, can I get a parking place, and is that shirt I saw last week at Macy’s still on the rack? Clocks will find the correct time after a power failure, the microwave
will download new recipes from the Internet, and kid’s toys will refresh themselves with new games and new vocabularies. Clothes labels will track washing, coffee cups will alert cleaning staff to mold, light switches will save energy if no one is in the room, and pencils will digitize everything we draw. Where will data mining be in this new world? Everywhere!

It’s hard to point to examples of a future that does not yet exist. But advances in user interface technology are suggestive. Many repetitive tasks in direct-manipulation computer interfaces cannot be automated with standard application tools, forcing computer users to perform the same interface actions repeatedly. This typifies the frustrations alluded to previously: who’s in charge—me or it? Experienced programmers might write a script to carry out such tasks on their behalf, but as operating systems accrue layer upon layer of complexity the power of programmers to command the machine is eroded and vanishes altogether when complex functionality is embedded in appliances rather than in general-purpose computers.

Research in programming by demonstration enables ordinary computer users to automate predictable tasks without requiring any programming knowledge at all. The user need only know how to perform the task in the usual way to be able to communicate it to the computer. One system, called Familiar, helps users automate iterative tasks involving existing applications on Macintosh computers. It works across applications and can work with completely new ones never before encountered. It does this by using Apple’s scripting language to glean information from each application and exploiting that information to make predictions. The agent tolerates noise. It generates explanations to inform the computer user about its predictions, and incorporates feedback. It’s adaptive: it learns specialized tasks for individual users. Furthermore, it is sensitive to each user’s style. If two people were teaching a task and happened to give identical demonstrations, Familiar would not necessarily infer identical programs—it’s tuned to their habits because it learns from their interaction history.

Familiar employs standard machine learning techniques to infer the user’s intent. Rules are used to evaluate predictions so that the best one can be presented to the user at each point. These rules are conditional so that users can teach classification tasks such as sorting files based on their type and assigning labels based on their size. They are learned incrementally: the agent adapts to individual users by recording their interaction history.

Many difficulties arise. One is scarcity of data. Users are loathe to demonstrate several iterations of a task—they think the agent should immediately catch on to what they are doing. Whereas a data miner would consider a 100-instance dataset miniscule, users bridle at the prospect of demonstrating a task even half a dozen times. A second difficulty is the plethora of attributes. The computer desktop environment has hundreds of features that any given action might depend upon. This means that small datasets are overwhelmingly likely
to contain attributes that are apparently highly predictive but nevertheless irrelevant, and specialized statistical tests are needed to compare alternative hypotheses. A third is that the iterative, improvement-driven development style that characterizes data mining applications fails. It is impossible \textit{in principle} to create a fixed training-and-testing corpus for an interactive problem such as programming by demonstration because each improvement in the agent alters the test data by affecting how users react to it. A fourth is that existing application programs provide limited access to application and user data: often the raw material on which successful operation depends is inaccessible, buried deep within the application program.

Data mining is already widely used at work. Text mining is starting to bring the techniques in this book into our own lives, as we read our email and surf the Web. As for the future, it will be stranger than we can imagine. The spreading computing infrastructure will offer untold opportunities for learning. Data mining will be there, behind the scenes, playing a role that will turn out to be foundational.

\section*{8.6 Further reading}

There is a substantial volume of literature that treats the topic of massive datasets, and we can only point to a few references here. Fayyad and Smith (1995) describe the application of data mining to voluminous data from scientific experiments. Shafer et al. (1996) describe a parallel version of a top-down decision tree inducer. A sequential decision tree algorithm for massive disk-resident datasets has been developed by Mehta et al. (1996). The technique of applying any algorithm to a large dataset by splitting it into smaller chunks and bagging or boosting the result is described by Breiman (1999); Frank et al. (2002) explain the related pruning and selection scheme.

Despite its importance, little seems to have been written about the general problem of incorporating metadata into practical data mining. A scheme for encoding domain knowledge into propositional rules and its use for both deduction and induction has been investigated by Giraud-Carrier (1996). The related area of inductive logic programming, which deals with knowledge represented by first-order logic rules, is covered by Bergadano and Gunetti (1996).

Text mining is an emerging area, and there are few comprehensive surveys of the area as a whole: Witten (2004) provides one. A large number of feature selection and machine learning techniques have been applied to text categorization (Sebastiani 2002). Martin (1995) describes applications of document clustering to information retrieval. Cavnar and Trenkle (1994) show how to use \textit{n}-gram profiles to ascertain with high accuracy the language in which a document is written. The use of support vector machines for authorship ascription is
described by Diederich et al. (2003); the same technology was used by Dumais et al. (1998) to assign key phrases from a controlled vocabulary to documents on the basis of a large number of training documents. The use of machine learning to extract key phrases from the document text has been investigated by Turney (1999) and Frank et al. (1999).

Appelt (1999) describes many problems of information extraction. Many authors have applied machine learning to seek rules that extract slot-fillers for templates, for example, Soderland et al. (1995), Huffman (1996), and Freitag (2002). Califf and Mooney (1999) and Nahm and Mooney (2000) investigated the problem of extracting information from job ads posted on Internet newsgroups. An approach to finding information in running text based on compression techniques has been reported by Witten et al. (1999). Mann (1993) notes the plethora of variations of Muammar Qaddafi on documents received by the Library of Congress.

Chakrabarti (2003) has written an excellent and comprehensive book on techniques of Web mining. Kushmerick et al. (1997) developed techniques of wrapper induction. The semantic Web was introduced by Tim Berners-Lee (Berners-Lee et al. 2001), who 10 years earlier developed the technology behind the World Wide Web.

The first paper on junk email filtering was written by Sahami et al. (1998). Our material on computer network security is culled from work by Yurcik et al. (2003). The information on the CAPPS system comes from the U.S. House of Representatives Subcommittee on Aviation (2002), and the use of unsupervised learning for threat detection is described by Bay and Schwabacher (2003). Problems with current privacy-preserving data mining techniques have been identified by Datta et al. (2003). Stone and Veloso (2000) surveyed multiagent systems of the kind that are used for playing robo-soccer from a machine learning perspective. The fascinating story of Ben Ish Chai and the technique used to unmask him is from Koppel and Schler (2004).

The vision of calm computing, as well as the examples we have mentioned, is from Weiser (1996) and Weiser and Brown (1997). More information on different methods of programming by demonstration can be found in compendia by Cypher (1993) and Lieberman (2001). Mitchell et al. (1994) report some experience with learning apprentices. Familiar is described by Paynter (2000). Permutation tests (Good 1994) are statistical tests that are suitable for small sample problems: Frank (2000) describes their application in machine learning.
Experience shows that no single machine learning scheme is appropriate to all data mining problems. The universal learner is an idealistic fantasy. As we have emphasized throughout this book, real datasets vary, and to obtain accurate models the bias of the learning algorithm must match the structure of the domain. Data mining is an experimental science.

The Weka workbench is a collection of state-of-the-art machine learning algorithms and data preprocessing tools. It includes virtually all the algorithms described in this 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, and 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—and even on a personal digital assistant. It provides a uniform interface to many different learning algorithms, along with methods for pre- and postprocessing and for evaluating the result of learning schemes on any given dataset.

9.1 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 described in Chapter 7. You can preprocess a dataset, feed it into a learning scheme, and analyze the resulting classifier and its performance—all without writing any program code at all.

The workbench includes methods for all the standard data mining problems: regression, classification, 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 in the ARFF format described in Section 2.4, which 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. The learning methods are called *classifiers*, and in the interactive Weka interface you select the one you want from a menu. Many classifiers 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 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. We will show how different filters can be used, list the filtering algorithms, and describe their parameters. Weka also includes implementations of algorithms for learning association rules, clustering data for which no class value is specified, and selecting relevant attributes in the data, which we describe briefly.
9.2 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 an ARFF file (or spreadsheet) and build a decision tree from it. But learning decision trees is just the beginning: there are many other algorithms to explore. The Explorer interface helps you do just that. It guides you by presenting choices as menus, by forcing you to work in an appropriate order by graying out options until they are applicable, and 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 obtain 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 two 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 immediately loads it all in. This 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 Experimenter to distribute the computing load across multiple machines using Java remote method invocation (RMI). In this way you can set up large-scale statistical experiments and leave them to run.

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 four different user interfaces: the Explorer, the Knowledge Flow, the Experimenter, and the command-line interface. We describe them in turn in the next chapters. Most people choose the Explorer, at least initially.

### 9.3 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 and how to identify Weka’s major building blocks, highlighting which parts contain supervised learning methods, which contain tools for data preprocessing, and which contain methods for other learning schemes. It 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 you how to do that by presenting an example of a simple data mining application in Java. This will enable you to become familiar with the basic data structures in Weka, representing instances, classifiers, and filters.

If you intend to become an expert in machine learning algorithms (or, indeed, if you already are one), you’ll probably want to implement your own algorithms without having to address such mundane details as reading the data from a file, implementing filtering algorithms, or providing code to evaluate the results. If so, we have good news for you: Weka already includes all this. 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.

### 9.4 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.
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 10.1. There are six different panels, selected by the tabs at the top, corresponding to the various data mining tasks that Weka supports.

### 10.1 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’s 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.
Preparing the data

The data is often presented in a spreadsheet or database. However, Weka’s native data storage method is ARFF format (Section 2.4). 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 (Figure 2.2). 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 10.2 shows an Excel spreadsheet containing the weather data from Section 1.2, the data in CSV form loaded into Microsoft Word, and the result of converting it manually into an ARFF file. However, you don’t 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.

Loading the data into the Explorer

Let’s load this data into the Explorer and start analyzing it. Fire up Weka to get the panel shown in Figure 10.3(a). Select Explorer from the four graphical user
interface choices at the bottom. (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 10.3(b). 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

Figure 10.2 Weather data: (a) spreadsheet, (b) CSV format, and (c) ARFF.
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 10.3(b). This tells you about the dataset: it has 14 instances and five attributes (center left); the attributes are called outlook, 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 10.3(b) 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 (an example appears in Figure 10.9 on page 384).

You can delete an attribute by clicking its checkbox and using the Remove button. All selects all the attributes, None selects none, and Invert inverts the current selection. 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.

Building a decision tree
To see what the C4.5 decision tree learner described in Section 6.1 does with this dataset, use the J4.8 algorithm, which is Weka’s implementation of this decision tree learner. (J4.8 actually implements a later and slightly improved version called C4.5 revision 8, which was the last public version of this family of algorithms before the commercial implementation C5.0 was released.) Click the Classify tab to get a screen that looks like Figure 10.4(b). 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 10.4(a), and finding J48. The menu structure represents the organization of the Weka code into modules, which will be described in Chapter 13. For now, just open up the hierarchy as necessary—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 10.4(b), 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 10.4(b) jumps up and dances—and then produces the output shown in the main panel of Figure 10.4(b).

Examining the output
Figure 10.5 shows the full output (Figure 10.4(b) only gives the lower half). At the beginning is a summary of the dataset, and the fact that tenfold cross-validation was used to evaluate it. That is the default, and if you look closely at Figure 10.4(b) you will see that the Cross-validation box at the left is checked. Then comes a pruned decision tree in textual form. 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
Figure 10.4 Using J4.8: (a) finding it in the classifiers list and (b) the Classify tab.
## Run information

**Scheme:**
```
weka.classifiers.trees.J48 -C 0.25 -M 2
```

**Relation:**
```
weather
```

**Instances:** 14

**Attributes:** 5

- Outlook
- 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

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correctly Classified Instances</td>
<td>9</td>
</tr>
<tr>
<td>Incorrectly Classified Instances</td>
<td>5</td>
</tr>
<tr>
<td>Kappa statistic</td>
<td>0.186</td>
</tr>
<tr>
<td>Mean absolute error</td>
<td>0.2857</td>
</tr>
<tr>
<td>Root mean squared error</td>
<td>0.4818</td>
</tr>
<tr>
<td>Relative absolute error</td>
<td>60%</td>
</tr>
<tr>
<td>Root relative squared error</td>
<td>97.6586%</td>
</tr>
<tr>
<td>Total Number of Instances</td>
<td>14</td>
</tr>
</tbody>
</table>

**Figure 10.5** Output from the J4.8 decision tree learner.
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: see pages 378–379 later in this chapter.

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 10.4(b). 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 (described in Section 5.7) 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 (Section 5.7), 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 quadratic loss (Section 5.6). 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 some statistics from page 172.
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 10.4(b)) 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 (Section 5.1). 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 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 10.4(b) 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 (Section 5.7), then edit the values as required.

The small pane at the lower left of Figure 10.4(b), 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.

Working with models

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 in 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 (page 324) and various cost and threshold curves (Section 5.7). For cost and threshold curves 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 (Table 5.7). 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 10.6 shows two ways of looking at the result of using J4.8 to classify the Iris dataset (Section 1.2)—we use this rather than the weather data because it produces more interesting pictures. Figure 10.6(a) 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 10.6(b) 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 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 10.6(b)). 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.

When things go wrong

Beneath the result history list, at the bottom of Figure 10.4(b), 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 weka in the lower right-hand corner of the interface.
Figure 10.6 Visualizing the result of J4.8 on the iris dataset: (a) the tree and (b) the classifier errors.
10.2 Exploring the Explorer

We have briefly investigated two of the six tabs at the top of the Explorer window in Figure 10.3(b) and Figure 10.4(b). 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 for the dataset.
4. **Associate**: Learn association rules for the data and evaluate them.
5. **Select attributes**: Select the most relevant aspects in 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 $\times$ shows how many concurrent processes that are running. If the bird is standing but stops moving, it’s sick! Something has gone wrong, and you should restart the Explorer.

**Loading and filtering files**

Along the top of the **Preprocess** panel in Figure 10.3(b) 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.

**Converting files to ARFF**

Weka has three file format converters: for spreadsheet files with the extension `.csv`, for C4.5’s native file format with the extensions `.names` and `.data`, and for serialized instances with the extension `.bsi`. The appropriate converter is used based on the 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 10.7(a).
This is a generic object editor, used throughout Weka for selecting and configuring objects. 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 10.7(b). It is always worth looking at the documentation! In this case, it explains that the spreadsheet’s first row determines the attribute names. Click OK to use this converter. For a different one, click Choose to select from the list in Figure 10.7(c).

The ArffLoader is the first option, and we reached this point only because it failed. The CSVLoader is the default, and we clicked Choose because we want a different one. The third 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 fourth, for serialized instances, is for reloading a dataset that has been saved as a Java serialized object. Any Java object can be saved in this form and reloaded. As a native Java format, it is quicker to load than an ARFF file, which must be parsed and checked. When repeatedly reloading a large dataset it may be worth saving it in this form.

Further features of the generic object editor in Figure 10.7(a) 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.)

Data can be saved in all these formats using the Save button in Figure 10.3(b). Apart from loading and saving datasets, the Preprocess panel also allows you to filter them. Filters are an important component of Weka.

**Using filters**

Clicking Choose (near the top left) in Figure 10.3(b) gives a list of filters like that in Figure 10.8(a). 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 10.4(b) to open the J4.8 classifier’s object editor.) The object editor for the Remove filter is shown in Figure 10.8(b). To learn about it, click More to show the information in Figure 10.8(c). 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 10.8(b), 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.

Apply the filter by clicking Apply (at the right-hand side of Figure 10.3(b)). Immediately the screen in Figure 10.9 appears—just like the one in Figure
10.3(b) but with only three attributes, humidity, windy, and play. At this point the fourth 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 distri-
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.

**Training and testing learning schemes**

The Classify panel lets you train and test learning schemes that perform classification or regression. Section 10.1 explained how to interpret the output of a decision tree learner and showed the performance figures that are automatically generated by the evaluation module. The interpretation of these is the same for all models that predict a categorical class. However, when evaluating models for numeric prediction, Weka produces a different set of performance measures.

As an example, in Figure 10.10(a) the CPU performance dataset from Table 1.5 (page 16) has been loaded into Weka. You can see the histogram of values of the first attribute, vendor, at the lower right. In Figure 10.10(b) 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 hierarchical menu shown in Figure 10.4(a), finding M5P, and clicking Start. The hierarchy helps to locate particular classifiers by grouping items with common functionality.
Figure 10.11 shows the output. The pruned model tree is simply a decision stump with a split on the $\text{MMAX}$ attribute and two linear models, one for each leaf. Both models involve a nominal attribute, $\text{vendor}$, as well as some numeric ones. The expression $\text{vendor} = \text{adviser, sperry, amdahl}$ is interpreted as follows: if $\text{vendor}$ is either $\text{adviser}$, $\text{sperry}$, or $\text{amdahl}$, then substitute 1; otherwise, substitute 0. The description of the model tree is followed by several figures that measure its performance. These are derived from the test option chosen in Figure 10.10(b), 10-fold cross-validation (not stratified, because that doesn’t
=== Run information ===

Scheme:       weka.classifiers.trees.M5P -M 4.0
Relation:     cpu
Instances:    209
Attributes:   8
  vendor
  MYCT
  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)

\[
\text{MMAX} \leq 14000 : \text{LM1} \ (141/4.178\%)
\]
\[
\text{MMAX} > 14000 : \text{LM2} \ (68/50.073\%)
\]

LM num: 1

class =
-2.0542 * vendor=honeywell,ipl,ibm,cdc,ncr,basf,gould,siemens,nas,adviser,sperry,amdahl
+ 5.4303 * vendor=adviser,sperry,amdahl
- 5.7791 * vendor=amdahl
+ 0.0064 * MYCT
+ 0.0016 * MMIN
+ 0.0034 * MMAX
+ 0.5524 * CACH
+ 1.1411 * CHMIN
+ 0.0945 * CHMAX
+ 4.1463

LM num: 2

class =
-57.3649 * vendor=honeywell,ipl,ibm,cdc,ncr,basf,gould,siemens,nas,adviser,sperry,amdahl
+ 46.1469 * vendor=adviser,sperry,amdahl
- 58.0762 * vendor=amdahl

Figure 10.11 Output from the M5' program for numeric prediction.
Ordinary linear regression (Section 4.6), another scheme for numeric prediction, is found under `LinearRegression` in the `functions` section of the menu in Figure 10.4(a). It builds a single linear regression model rather than the two in Figure 10.11; not surprisingly, its performance is slightly worse.

To get a feel for their relative performance, let's visualize the errors these schemes make, as we did for the Iris dataset in Figure 10.6(b). 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 10.12. The points are color coded by class—but in this case the color varies continuously because the class is numeric. In Figure 10.12 the `Vendor` attribute has been selected for the X-axis and the instance number has been chosen for the Y-axis because this gives a good 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 10.12(a) (for M5'), when compared with those in Figure 10.12(b) (for linear regression), show that M5' is superior.

\[
\begin{align*}
+ 0.012 \times \text{MYCT} \\
+ 0.0162 \times \text{MMIN} \\
+ 0.0086 \times \text{MMAX} \\
+ 0.8332 \times \text{CACH} \\
- 1.2665 \times \text{CHMIN} \\
+ 1.2741 \times \text{CHMAX} \\
- 107.243
\end{align*}
\]

Number of Rules : 2

Time taken to build model: 1.37 seconds

### Cross-validation

### Summary

- Correlation coefficient: 0.9766
- Mean absolute error: 13.6917
- Root mean squared error: 35.3003
- Relative absolute error: 15.6194 %
- Root relative squared error: 22.8092 %
- Total Number of Instances: 209
Do it yourself: The User Classifier

The User Classifier (mentioned at the end of Section 3.2) allows Weka users to build their own classifiers interactively. It resides in the trees section of the hierarchical menu in Figure 10.4(a) under UserClassifier. We illustrate its operation on a new problem, segmenting visual image data into classes such as grass, sky, foliage, brick, and cement based on attributes giving average intensity, hue, size,
position, and various simple textural features. The training data file is supplied with the Weka distribution and called *segment-challenge.arff*. Having loaded it in, select the User Classifier. For evaluation use the special test set called *segment-test.arff* as the *Supplied test set* on the *Classify* panel. Evaluation by cross-validation is impossible when you have to construct a classifier manually for each fold.

Following *Start*, a new window appears and Weka waits for you to build the classifier. The *Tree Visualizer* and *Data Visualizer* tabs switch between different views. The former shows the current state of the classification tree, and each node gives the number of instances of each class at that node. The aim is to come up with a tree in which the leaf nodes are as pure as possible. Initially there is only one node, the root, which contains all the data. Switch to the *Data Visualizer* to create a split. This shows the same two-dimensional plot that we saw in Figure 10.6(b) for the Iris dataset and Figure 10.12 for the CPU performance data. The attributes to use for X and Y are selected as before, and the goal here is to find a combination that separates the classes as cleanly as possible. Figure 10.13(a) shows a good choice: *region–centroid–row* for X and *intensity–mean* for Y.

Having found a good separation, you must specify a region in the graph. Four tools for this appear in the pull-down menu below the Y-axis selector. *Select Instance* identifies a particular instance. *Rectangle* (shown in Figure 10.13(a)) allows you to drag out a rectangle on the graph. With *Polygon* and *Polyline* you build a free-form polygon or draw a free-form polyline (left-click to add a vertex and right-click to complete the operation). Once an area has been selected, it turns gray. In Figure 10.13(a) the user has defined a rectangle. The *Submit* button creates two new nodes in the tree, one holding the selected instances and the other with all the rest. *Clear* clears the selection; *Save* saves the instances in the current tree node as an ARFF file.

At this point, the *Tree Visualizer* shows the tree in Figure 10.13(b). There is a pure node for the *sky* class, but the other node is mixed and should be split further. Clicking on different nodes determines which subset of data is shown by the *Data Visualizer*. Continue adding nodes until you are satisfied with the result—that is, until the leaf nodes are mostly pure. Then right-click on any blank space in the *Tree Visualizer* and choose *Accept the Tree*. Weka evaluates your tree on the test set and outputs performance statistics (80% is a good score on this problem).

Building trees manually is very tedious. But Weka can complete the task for you by building a subtree under any node: just right-click the node.

### Using a metalearner

Metalearners (Section 7.5) take simple classifiers and turn them into more powerful learners. For example, to boost decision stumps in the Explorer, go to the
Figure 10.13 Working on the segmentation data with the User Classifier: (a) the data visualizer and (b) the tree visualizer.
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 10.14 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.

**Clustering and association rules**

Use the Cluster and Associate panels to invoke clustering algorithms (Section 6.6) and methods for finding association rules (Section 4.5). 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 10.6(b). 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 only three algorithms for determining association rules and no methods for evaluating such rules. Figure 10.15 shows the output from the Apriori program for association rules (described in Section 4.5) on the nominal version of the weather 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 in 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.

**Attribute selection**

The Select attributes panel gives access to several methods for attribute selection. As explained in Section 7.1, this involves an attribute evaluator and a search

<table>
<thead>
<tr>
<th>Rule Number</th>
<th>Antecedent</th>
<th>Consequent</th>
<th>Support</th>
<th>Confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>outlook=overcast</td>
<td>play=yes 4</td>
<td>4</td>
<td>conf:(1)</td>
</tr>
<tr>
<td>2.</td>
<td>temperature=cool</td>
<td>humidity=normal</td>
<td>4</td>
<td>conf:(1)</td>
</tr>
<tr>
<td>3.</td>
<td>humidity=normal windy=FALSE</td>
<td>play=yes 4</td>
<td>4</td>
<td>conf:(1)</td>
</tr>
<tr>
<td>4.</td>
<td>outlook=sunny play=no 3</td>
<td>humidity=high</td>
<td>3</td>
<td>conf:(1)</td>
</tr>
<tr>
<td>5.</td>
<td>outlook=sunny humidity=high 3</td>
<td>play=no 3</td>
<td>3</td>
<td>conf:(1)</td>
</tr>
<tr>
<td>6.</td>
<td>outlook=rainy play=yes 3</td>
<td>windy=FALSE 3</td>
<td>3</td>
<td>conf:(1)</td>
</tr>
<tr>
<td>7.</td>
<td>outlook=rainy windy=FALSE 3</td>
<td>play=yes 3</td>
<td>3</td>
<td>conf:(1)</td>
</tr>
<tr>
<td>8.</td>
<td>temperature=cool play=yes 3</td>
<td>humidity=normal</td>
<td>3</td>
<td>conf:(1)</td>
</tr>
<tr>
<td>9.</td>
<td>outlook=sunny temperature=hot 2</td>
<td>humidity=high 2</td>
<td>2</td>
<td>conf:(1)</td>
</tr>
<tr>
<td>10.</td>
<td>temperature=hot play=no 2</td>
<td>outlook=sunny 2</td>
<td>2</td>
<td>conf:(1)</td>
</tr>
</tbody>
</table>

**Figure 10.15** Output from the Apriori program for association rules.
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).

**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 10.16(a) 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, 1000 instances at the same data point would look just the same as 1 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 10.16(b). 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.

### 10.3 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 in Chapters 11 and 12. 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 (Section 7.2). This seemingly innocuous distinction 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. When using supervised filters you must 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.

Figure 10.16 Visualizing the Iris dataset.
and look at its associated object editor, which defines what the filter does and the parameters it takes.

**Unsupervised attribute filters**

Table 10.1 lists Weka’s unsupervised attribute filters. Many of the operations were introduced in Section 7.3.

**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). 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 1–3 for the first three attributes,
<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Add</strong></td>
<td>Add a new attribute, whose values are all marked as <em>missing</em>.</td>
</tr>
<tr>
<td><strong>AddCluster</strong></td>
<td>Add a new nominal attribute representing the cluster assigned to each</td>
</tr>
<tr>
<td></td>
<td>instance by a given clustering algorithm.</td>
</tr>
<tr>
<td><strong>AddExpression</strong></td>
<td>Create a new attribute by applying a specified mathematical function to</td>
</tr>
<tr>
<td></td>
<td>existing attributes.</td>
</tr>
<tr>
<td><strong>AddNoise</strong></td>
<td>Change a percentage of a given nominal attribute’s values.</td>
</tr>
<tr>
<td><strong>ClusterMembership</strong></td>
<td>Use a clusterer to generate cluster membership values, which then form</td>
</tr>
<tr>
<td></td>
<td>the new attributes.</td>
</tr>
<tr>
<td><strong>Copy</strong></td>
<td>Copy a range of attributes in the dataset.</td>
</tr>
<tr>
<td><strong>Discretize</strong></td>
<td>Convert numeric attributes to nominal: Specify which attributes, number</td>
</tr>
<tr>
<td></td>
<td>of bins, whether to optimize the number of bins, and output binary</td>
</tr>
<tr>
<td></td>
<td>attributes. Use equal-width (default) or equal-frequency binning.</td>
</tr>
<tr>
<td><strong>FirstOrder</strong></td>
<td>Apply a first-order differencing operator to a range of numeric attributes.</td>
</tr>
<tr>
<td><strong>MakeIndicator</strong></td>
<td>Replace a nominal attribute with a Boolean attribute. Assign value 1 to</td>
</tr>
<tr>
<td></td>
<td>instances with a particular range of attribute values; otherwise, assign 0.</td>
</tr>
<tr>
<td></td>
<td>By default, the Boolean attribute is coded as numeric.</td>
</tr>
<tr>
<td><strong>MergeTwoValues</strong></td>
<td>Merge two values of a given attribute: Specify the index of the two values to be merged.</td>
</tr>
<tr>
<td><strong>NominalToBinary</strong></td>
<td>Change a nominal attribute to several binary ones, one for each value.</td>
</tr>
<tr>
<td><strong>Normalize</strong></td>
<td>Scale all numeric values in the dataset to lie within the interval [0,1].</td>
</tr>
<tr>
<td><strong>NumericToBinary</strong></td>
<td>Convert all numeric attributes into binary ones: Nonzero values become 1.</td>
</tr>
<tr>
<td><strong>NumericTransform</strong></td>
<td>Transform a numeric attribute using any Java function.</td>
</tr>
<tr>
<td><strong>Obfuscate</strong></td>
<td>Obfuscate the dataset by renaming the relation, all attribute names, and</td>
</tr>
<tr>
<td></td>
<td>nominal and string attribute values.</td>
</tr>
<tr>
<td><strong>PKIDiscretize</strong></td>
<td>Discretize numeric attributes using equal-frequency binning, where the</td>
</tr>
<tr>
<td></td>
<td>number of bins is equal to the square root of the number of values</td>
</tr>
<tr>
<td></td>
<td>(excluding missing values).</td>
</tr>
<tr>
<td><strong>RandomProjection</strong></td>
<td>Project the data onto a lower-dimensional subspace using a random matrix.</td>
</tr>
<tr>
<td><strong>Remove</strong></td>
<td>Remove attributes.</td>
</tr>
<tr>
<td><strong>RemoveType</strong></td>
<td>Remove attributes of a given type (nominal, numeric, string, or date).</td>
</tr>
<tr>
<td><strong>RemoveUseless</strong></td>
<td>Remove constant attributes, along with nominal attributes that vary too</td>
</tr>
<tr>
<td></td>
<td>much.</td>
</tr>
<tr>
<td><strong>ReplaceMissingValues</strong></td>
<td>Replace all missing values for nominal and numeric attributes with the modes and means of the training data.</td>
</tr>
<tr>
<td><strong>Standardize</strong></td>
<td>Standardize all numeric attributes to have zero mean and unit variance.</td>
</tr>
<tr>
<td><strong>StringToNominal</strong></td>
<td>Convert a string attribute to nominal.</td>
</tr>
<tr>
<td><strong>StringToWordVector</strong></td>
<td>Convert a string attribute to a vector that represents word occurrence frequencies; you can choose the delimiter(s)—and there are many more options.</td>
</tr>
<tr>
<td><strong>SwapValues</strong></td>
<td>Swap two values of an attribute.</td>
</tr>
<tr>
<td><strong>TimeSeriesDelta</strong></td>
<td>Replace attribute values in the current instance with the difference between the current value and the value in some previous (or future) instance.</td>
</tr>
<tr>
<td><strong>TimeSeriesTranslate</strong></td>
<td>Replace attribute values in the current instance with the equivalent value in some previous (or future) instance.</td>
</tr>
</tbody>
</table>
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.

Remove has already been described. Similar filters are RemoveType, which
deletes all attributes of a given type (nominal, numeric, string, or date), 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. Some unsupervised attrib-
ute filters behave differently if the menu in the Preprocess panel has been used
to set a class attribute. For example, RemoveType and RemoveUseless both skip
the class attribute.

AddCluster applies a clustering algorithm to the data before filtering it. You
use the object editor to choose the clustering algorithm. Clusterers are config-
ured just as filters are (Section 10.6). The 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 in a book! At any rate,
one 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 left unaltered.

AddExpression creates a new attribute by applying a mathematical function
to numeric attributes. The expression can contain attribute references and con-
stants; the arithmetic operators +, −, *, /, and ^; the functions log and exp, abs
and sqrt, floor, ceil and rint,\(^5\) and sin, cos, and tan; and parentheses. Attributes
are specified by the prefix a, for example, a7 is the seventh attribute. An example
expression is

\[
a1^2 * a5 / log(a7 * 4.0)
\]

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

Whereas AddExpression applies mathematical functions, 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.

\(^5\) The rint function rounds to the closest integer.
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. Standardize transforms them to have zero mean and unit variance. Both skip the class attribute, if set.

**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.

One way of dealing with missing values is to replace them globally before applying a learning scheme. ReplaceMissingValues replaces each missing value with the mean for numeric attributes and the mode for nominal ones. If a class is set, missing values of that attribute are not replaced.

**Conversions**

Many filters convert attributes from one form to another. Discretize uses equal-width or equal-frequency binning (Section 7.2) 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. PKIDiscretize discretizes numeric attributes using equal-frequency binning in which the number of bins is the square root of the number of values (excluding missing values). Both these filters skip the class attribute.

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 value for each instance is 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.

Some learning schemes, such as support vector machines, only handle binary attributes. The NominalToBinary filter transforms all multivalued nominal attributes.
attributes in a dataset into binary ones, replacing each attribute with $k$ values by $k$ binary attributes using a simple one-per-value encoding. 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 attribute will be missing; otherwise, the value of the new attribute will be 1. These filters also skip the class attribute.

*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 $-1$.

**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.

*StringToWordVector* produces attributes that represent the frequency of each word in the string. The set of words—that is, the new attribute set—is determined from the dataset. By default each word becomes an attribute whose value is 1 or 0, reflecting that word’s presence in the string. 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. They can be converted to lowercase before being added to the dictionary, or all words on a predetermined list of English 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. 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 to be the average length of all documents that appear as values of the original string attribute. Alternatively, the frequencies $f_{ij}$ for word $i$ in document $j$ can be transformed using $\log (1 + f_{ij})$ or the TF $\times$ IDF measure (Section 7.3).

**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. \textit{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 may be used.

\textbf{Randomizing}

Other attribute filters degrade the data. \textit{AddNoise} takes a nominal attribute and changes a given percentage of its values. Missing values can be retained or changed along with the rest. \textit{Obfuscate} anonymizes data by renaming the relation, attribute names, and nominal and string attribute values. \textit{RandomProjection} projects the dataset on to a lower-dimensional subspace using a random matrix with columns of unit length (Section 7.3). The class attribute is not included in the projection.

\textbf{Unsupervised instance filters}

Weka’s instance filters, listed in Table 10.2, affect all instances in a dataset rather than all values of a particular attribute or attributes.

\textbf{Randomizing and subsampling}

You can \textit{Randomize} the order of instances in the dataset. \textit{Normalize} treats all numeric attributes (excluding the class) as a vector and normalizes it to a given length. You can specify the vector length and the norm to be used.

There are various ways of generating subsets of the data. Use \textit{Resample} to produce a random sample by sampling with replacement or \textit{RemoveFolds} to split

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Name} & \textbf{Function} \tabularnewline
\hline
\textit{NonSparseToSparse} & Convert all incoming instances to sparse format (Section 2.4) \tabularnewline
\textit{Normalize} & Treat numeric attributes as a vector and normalize it to a given length \tabularnewline
\textit{Randomize} & Randomize the order of instances in a dataset \tabularnewline
\textit{RemoveFolds} & Output a specified cross-validation fold for the dataset \tabularnewline
\textit{RemoveMisclassified} & Remove instances incorrectly classified according to a specified classifier—useful for removing outliers \tabularnewline
\textit{RemovePercentage} & Remove a given percentage of a dataset \tabularnewline
\textit{RemoveRange} & Remove a given range of instances from a dataset \tabularnewline
\textit{RemoveWithValues} & Filter out instances with certain attribute values \tabularnewline
\textit{Resample} & Produce a random subsample of a dataset, sampling with replacement \tabularnewline
\textit{SparseToNonSparse} & Convert all incoming sparse instances into nonsparse format \tabularnewline
\hline
\end{tabular}
\end{table}
it into a given number of cross-validation folds and reduce it to just one of them. If a random number seed is provided, the dataset will be shuffled before the subset is extracted. 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.

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.

**Sparse instances**
The NonSparseToSparse and SparseToNonSparse filters convert between the regular representation of a dataset and its sparse representation (see Section 2.4).

**Supervised filters**
Supervised filters are available from the Explorer’s Preprocess panel, just as unsupervised ones are. You need to be careful with them because, despite appearances, they are not really preprocessing operations. We noted this previously with regard to discretization—the test data splits must not use the test data’s class values because these are supposed to be unknown—and it 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 meta-learner is provided that invokes a filter in a way that wraps the learning algorithm into the filtering mechanism. This filters the test data using the filter that has been created by 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: 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 in the meta section of the menu displayed by the Classify panel’s Choose button. Figure 10.17(a) shows the object editor for this metalearning scheme. With it you choose a classifier and a filter. Figure 10.17(b) shows the menu of filters.

Supervised filters, like unsupervised ones, are divided into attribute and instance filters, listed in Table 10.3 and Table 10.4.
Supervised attribute filters

Discretize, highlighted in Figure 10.17, uses the MDL method of supervised discretization (Section 7.2). 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.

Table 10.3  Supervised attribute filters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>AttributeSelection</td>
<td>Provides access to the same attribute selection methods as the</td>
</tr>
<tr>
<td></td>
<td>Select attributes panel</td>
</tr>
<tr>
<td>ClassOrder</td>
<td>Randomize, or otherwise alter, the ordering of class values</td>
</tr>
<tr>
<td>Discretize</td>
<td>Convert numeric attributes to nominal</td>
</tr>
<tr>
<td>NominalToBinary</td>
<td>Convert nominal attributes to binary, using a supervised method</td>
</tr>
<tr>
<td></td>
<td>if the class is numeric</td>
</tr>
</tbody>
</table>

Table 10.4  Supervised instance filters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resample</td>
<td>Produce a random subsample of a dataset, sampling with replacement</td>
</tr>
<tr>
<td>SpreadSubsample</td>
<td>Produce a random subsample with a given spread between class frequencies,</td>
</tr>
<tr>
<td></td>
<td>sampling with replacement</td>
</tr>
<tr>
<td>StratifiedRemoveFolds</td>
<td>Output a specified stratified cross-validation fold for the dataset</td>
</tr>
</tbody>
</table>

Supervised attribute filters

Discretize, highlighted in Figure 10.17, uses the MDL method of supervised discretization (Section 7.2). 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 Section 6.5 (page 246) 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 order of class frequency. This filter must not be used with the **FilteredClassifier** meta-learning scheme! **AttributeSelection** can be used for automatic attribute selection and provides the same functionality as the Explorer’s *Select attributes* panel (described later).

### Supervised instance filters

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

## 10.4 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 parameters specified with minus signs. To change them, click that line to get an appropriate object editor. Table 10.5 lists Weka’s classifiers. They 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.

### Bayesian classifiers

**NaiveBayes** implements the probabilistic Naïve Bayes classifier (Section 4.2). **NaiveBayesSimple** uses the normal distribution to model numeric attributes. **NaiveBayes** can use kernel density estimators, which improves performance if the normality assumption is grossly incorrect; it can also handle numeric
Table 10.5  Classifier algorithms in Weka.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bayes</td>
<td></td>
</tr>
<tr>
<td>BayesNet</td>
<td>Learn Bayesian nets</td>
</tr>
<tr>
<td>ComplementNaiveBayes</td>
<td>Build a Complement Naïve Bayes classifier</td>
</tr>
<tr>
<td>NaiveBayes</td>
<td>Standard probabilistic Naïve Bayes classifier</td>
</tr>
<tr>
<td>NaiveBayesMultinomial</td>
<td>Multinomial version of Naïve Bayes</td>
</tr>
<tr>
<td>NaiveBayesSimple</td>
<td>Simple implementation of Naïve Bayes</td>
</tr>
<tr>
<td>NaiveBayesUpdateable</td>
<td>Incremental Naïve Bayes classifier that learns one instance at a time</td>
</tr>
<tr>
<td>Trees</td>
<td></td>
</tr>
<tr>
<td>ADTree</td>
<td>Build alternating decision trees</td>
</tr>
<tr>
<td>DecisionStump</td>
<td>Build one-level decision trees</td>
</tr>
<tr>
<td>Id3</td>
<td>Basic divide-and-conquer decision tree algorithm</td>
</tr>
<tr>
<td>J48</td>
<td>C4.5 decision tree learner (implements C4.5 revision 8)</td>
</tr>
<tr>
<td>LMT</td>
<td>Build logistic model trees</td>
</tr>
<tr>
<td>M5P</td>
<td>M5’ model tree learner</td>
</tr>
<tr>
<td>NBTree</td>
<td>Build a decision tree with Naïve Bayes classifiers at the leaves</td>
</tr>
<tr>
<td>RandomForest</td>
<td>Construct random forests</td>
</tr>
<tr>
<td>RandomTree</td>
<td>Construct a tree that considers a given number of random features at each node</td>
</tr>
<tr>
<td>REPTree</td>
<td>Fast tree learner that uses reduced-error pruning</td>
</tr>
<tr>
<td>UserClassifier</td>
<td>Allow users to build their own decision tree</td>
</tr>
<tr>
<td>Rules</td>
<td></td>
</tr>
<tr>
<td>ConjunctiveRule</td>
<td>Simple conjunctive rule learner</td>
</tr>
<tr>
<td>DecisionTable</td>
<td>Build a simple decision table majority classifier</td>
</tr>
<tr>
<td>JRip</td>
<td>RIPPER algorithm for fast, effective rule induction</td>
</tr>
<tr>
<td>M5Rules</td>
<td>Obtain rules from model trees built using M5’</td>
</tr>
<tr>
<td>Nnge</td>
<td>Nearest-neighbor method of generating rules using nonnested generalized exemplars</td>
</tr>
<tr>
<td>OneR</td>
<td>1R classifier</td>
</tr>
<tr>
<td>Part</td>
<td>Obtain rules from partial decision trees built using J4.8</td>
</tr>
<tr>
<td>Prism</td>
<td>Simple covering algorithm for rules</td>
</tr>
<tr>
<td>Ridor</td>
<td>Ripple-down rule learner</td>
</tr>
<tr>
<td>ZeroR</td>
<td>Predict the majority class (if nominal) or the average value (if numeric)</td>
</tr>
<tr>
<td>Functions</td>
<td></td>
</tr>
<tr>
<td>LeastMedSq</td>
<td>Robust regression using the median rather than the mean</td>
</tr>
<tr>
<td>LinearRegression</td>
<td>Standard linear regression</td>
</tr>
<tr>
<td>Logistic</td>
<td>Build linear logistic regression models</td>
</tr>
<tr>
<td>MultilayerPerceptron</td>
<td>Backpropagation neural network</td>
</tr>
<tr>
<td>PaceRegression</td>
<td>Build linear regression models using Pace regression</td>
</tr>
<tr>
<td>RBFNetwork</td>
<td>Implements a radial basis function network</td>
</tr>
<tr>
<td>SimpleLinearRegression</td>
<td>Learn a linear regression model based on a single attribute</td>
</tr>
<tr>
<td>SimpleLogistic</td>
<td>Build linear logistic regression models with built-in attribute selection</td>
</tr>
<tr>
<td>SMO</td>
<td>Sequential minimal optimization algorithm for support vector classification</td>
</tr>
</tbody>
</table>
attributes using supervised discretization. *NaiveBayesUpdateable* is an incremental version that processes one instance at a time; it can use a kernel estimator but not discretization. *NaiveBayesMultinomial* implements the multinomial Bayes classifier (Section 4.2, page 95). *ComplementNaiveBayes* builds a Complement Naïve Bayes classifier as described by Rennie et al. (2003) (the TF ¥ IDF and length normalization transforms used in this paper can be performed using the StringToWordVector filter).

*AODE* (averaged, one-dependence estimators) is a Bayesian method that averages over a space of alternative Bayesian models that have weaker independence assumptions than Naïve Bayes (Webb et al., 2005). The algorithm may yield more accurate classification than Naïve Bayes on datasets with nonindependent attributes.

*BayesNet* learns Bayesian networks 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 two different algorithms for estimating the conditional probability tables of the network. Search is done using K2 or the TAN algorithm (Section 6.7) or more sophisticated methods based on hill-climbing, simulated annealing, tabu search, and genetic algorithms. Optionally, search speed can be improved using AD trees (Section 6.7). There is also an algorithm that uses conditional independence tests to learn the structure of the network; alternatively, the network structure can be loaded from an XML (extensible markup language) 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 *Visualize graph*. Figure 10.18(a) shows the graph for the nominal version of the weather data, which in fact corresponds to the Naïve Bayes result.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SMOreg</strong></td>
<td>Sequential minimal optimization algorithm for support vector regression</td>
</tr>
<tr>
<td><em>VotedPerceptron</em></td>
<td>Voted perceptron algorithm</td>
</tr>
<tr>
<td><em>Winnow</em></td>
<td>Mistake-driven perceptron with multiplicative updates</td>
</tr>
<tr>
<td><strong>Lazy</strong></td>
<td></td>
</tr>
<tr>
<td><em>IB1</em></td>
<td>Basic nearest-neighbor instance-based learner</td>
</tr>
<tr>
<td><em>IBk</em></td>
<td>k-nearest-neighbor classifier</td>
</tr>
<tr>
<td><em>KStar</em></td>
<td>Nearest neighbor with generalized distance function</td>
</tr>
<tr>
<td><em>LBR</em></td>
<td>Lazy Bayesian Rules classifier</td>
</tr>
<tr>
<td><em>LWL</em></td>
<td>General algorithm for locally weighted learning</td>
</tr>
<tr>
<td><strong>Misc.</strong></td>
<td></td>
</tr>
<tr>
<td><em>Hyperpipes</em></td>
<td>Extremely simple, fast learner based on hypervolumes in instance space</td>
</tr>
<tr>
<td><em>VFI</em></td>
<td>Voting feature intervals method, simple and fast</td>
</tr>
</tbody>
</table>
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 10.18(b). Clicking on a node shows its probability distribution—Figure 10.18(c) is obtained by clicking on the windy node in (b).

Trees

Of the tree classifiers in Table 10.5 we have already seen how to use J4.8, which reimplements C4.5 (Section 6.1). To see the options, click the line beside the Choose button in Figure 10.4(b) to bring up the object editor in Figure 10.19. 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 (Section 6.2). 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 (pages 377–378) 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 (Section 4.2).

Table 10.5 shows many other decision tree methods. Id3 is the basic algorithm explained in Chapter 4. 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 RandomForest constructs random forests by bagging ensembles of random trees (Section 7.5, pages 320–321).

REPTree builds a decision or regression tree using information gain/variance reduction and prunes it using reduced-error pruning (Section 6.2, page 203). Optimized for speed, it only sorts values for numeric attributes once
(Section 6.1, page 190). 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.

**NBTree** is a hybrid between decision trees and Naïve Bayes. It creates trees whose leaves are Naïve Bayes classifiers for the instances that reach the leaf. When constructing the tree, cross-validation is used to decide whether a node should be split further or a Naïve Bayes model should be used instead (Kohavi 1996).

**M5P** is the model tree learner described in Section 6.5. **LMT** builds logistic model trees (Section 7.5, page 331). **LMT** can deal with binary and multiclass target variables, numeric and nominal attributes, and missing values. When fitting the logistic regression functions at a node, 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. Normally, it is the misclassification error that cross-validation minimizes, but the root mean-squared error of the probabilities 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.

**ADTree** builds an alternating decision tree using boosting (Section 7.5, pages 329–331) and is optimized for two-class problems. The number of boosting iterations is a parameter that can be tuned to suit the dataset and the desired complexity–accuracy tradeoff. Each iteration adds three nodes to the tree (one split node and two prediction nodes) unless nodes can be merged. The default search method is exhaustive search (Expand all paths); the others are heuristics and are much faster. You can determine whether to save instance data for visualization.

**Rules**

Table 10.5 shows many methods for generating rules. **DecisionTable** builds a decision table majority classifier (Section 7.1, page 295). 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 (Section 4.1) with one parameter: the minimum bucket size for discretization. **ConjunctiveRule** learns a single rule that predicts either a numeric or a nominal class value. Uncovered test instances are assigned the default class.
value (or distribution) of the uncovered training instances. The information gain (nominal class) or variance reduction (numeric class) of each antecedent is computed, and rules are pruned using reduced-error pruning. ZeroR is even simpler: it predicts the test data’s majority class (if nominal) or average value (if numeric). Prism implements the elementary covering algorithm for rules (Section 4.4).

Part obtains rules from partial decision trees (Section 6.2, pages 207–210). It builds the tree using C4.5’s heuristics with the same user-defined parameters as J4.8. M5Rules obtains regression rules from model trees built using M5’ (Section 6.5, pages 250–251). Ridor learns rules with exceptions (Section 6.2, pages 210–213) by generating the default rule, using incremental reduced-error pruning to find exceptions with the smallest error rate, finding the best exceptions for each exception, and iterating.

JRip implements RIPPER (Section 6.2, pages 205–207), including heuristic global optimization of the rule set (Cohen 1995). Ngne is a nearest-neighbor method for generating rules using nonnested generalized exemplars (Section 6.4, pages 238–239).

**Functions**

The functions category of Table 10.5 includes 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: Naïve Bayes has a simple mathematical formulation). Three of them implement linear regression (Section 4.6). 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. LinearRegression performs standard least-squares linear regression and can optionally perform attribute selection, either by greedily using backward elimination (Section 7.1) or by building a full model from all attributes and dropping terms one by one in decreasing order of their standardized coefficients until a stopping criteria is reached (this method was described in a slightly different context in Section 6.5 under Pruning the tree, page 245). Both methods use a version of the AIC termination criterion of Section 6.7 (page 277). The implementation has two further refinements: a mechanism 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, which is described in standard statistics texts.

LeastMedSq is a robust linear regression method that minimizes the median (rather than the mean) of the squares of divergences from the regression line (Section 7.4) (Rousseeuw and Leroy 1987). It repeatedly applies standard linear
regression to subsamples of the data and outputs the solution that has the smallest median-squared error.

SMO implements the sequential minimal optimization algorithm for training a support vector classifier (Section 6.3), using polynomial or Gaussian kernels (Platt 1998, Keerthi et al. 2001). 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 can be standardized to zero mean and unit variance. Pairwise classification is used for multiclass problems. Logistic regression models can be fitted to the support vector machine output to obtain probability estimates. In the multiclass case the predicted probabilities will be coupled pairwise (Hastie and Tibshirani 1998). When working with sparse instances, turn normalization off for faster operation. SMOreg implements the sequential minimal optimization algorithm for regression problems (Smola and Schölkopf 1998).

VotedPerceptron is the voted perceptron algorithm (Section 6.3, pages 222–223). Winnow (Section 4.6, pages 126–128) modifies the basic perceptron to use multiplicative updates. The implementation allows for a second multiplier, $\beta$—different from $1/\alpha$—to be used in place of the divisions in Figure 4.11, and also provides the balanced version of the algorithm.

PaceRegression builds linear regression models using the new technique of Pace regression (Wang and Witten 2002). When there are many attributes, Pace regression is particularly good at determining which ones to discard—which under certain regularity conditions it is provably optimal as the number of attributes tends to infinity.

SimpleLogistic builds logistic regression models (Section 4.6, pages 121–124), 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 (Landwehr et al. 2003). Logistic is an alternative implementation for building and using a multinomial logistic regression model with a ridge estimator to guard against overfitting by penalizing large coefficients, based on work by le Cessie and van Houwelingen (1992).

RBFNetwork implements a Gaussian radial basis function network (Section 6.3, page 234), deriving the centers and widths of hidden units using $k$-means and combining the outputs obtained from the hidden layer using logistic regression if the class is nominal and linear regression if it is numeric. The activations of the basis functions are normalized to sum to one before they are fed into the linear models. You can specify $k$, the number of clusters; the maximum number of logistic regression iterations for nominal-class problems; the minimum standard deviation for the clusters; and the ridge value for regression. If the class is nominal, $k$-means is applied separately to each class to derive $k$ clusters for each class.
Neural networks

*MultilayerPerceptron* is a neural network that trains using backpropagation (Section 6.3, page 227). Although listed under functions in Table 10.5, 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 10.20 appears in a separate window. This network has three layers: an input layer on the left with one rectangular box for each

![Diagram](image)

(a)

![Diagram](image)

(b)

**Figure 10.20** Using Weka’s neural-network graphical user interface.
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 10.20(a) 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 10.20(a), 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
10.20(b). 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 learn-
ing rate, its momentum (Section 6.3), and the number of passes it will take
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 10.20. 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 con-
tinue 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
10.20(a), but without autoBuild this would not appear and there would be no
connections. The hiddenLayers parameter defines the hidden layers present and
how many nodes each one contains. Figure 10.20(a) is generated by a value of
4 (one hidden layer with four nodes), and although Figure 10.20(b) was created
by adding nodes interactively, it could have been generated by setting hidden-
Layers 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 10.20(a).

The parameters \( \text{learningRate} \) and \( \text{Momentum} \) set values for these variables, which can be overridden in the graphical interface. A \( \text{decay} \) 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 \( \text{reset} \) 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 user interface is not used).

The \( \text{trainingTime} \) parameter sets the number of training epochs. Alternatively, a percentage of the data can be set aside for validation (using \( \text{validationSetSize} \)): 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 \( \text{validationThreshold} \) parameter determines how many consecutive times the validation set error can deteriorate before training is stopped.

The \( \text{nominalToBinaryFilter} \) filter is specified by default in the \( \text{MultilayerPerceptron} \) object editor; turning it off may improve performance on data in which the nominal attributes are really ordinal. The attributes can be normalized (with \( \text{normalizeAttributes} \)), and a numeric class can be normalized too (with \( \text{normalizeNumericClass} \)): both may improve performance.

**Lazy classifiers**

Lazy learners store the training instances and do no real work until classification time. \( \text{IB1} \) is a basic instance-based learner (Section 4.7) which finds the training instance closest in Euclidean distance to the given test instance and predicts the same class as this training instance. If several instances qualify as the closest, the first one found is used. \( \text{IBk} \) is a \( k \)-nearest-neighbor classifier that uses the same distance metric. 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. \( \text{KStar} \) is a nearest-neighbor method with a generalized distance function based on transformations (Section 6.4, pages 241–242).
**LBR** (for *Lazy Bayesian Rules*) is a Bayesian classifier that defers all processing to classification time. For each test instance it selects a set of attributes for which the independence assumption should not be made; the others are treated as independent of each other given the class and the selected set of attributes. It works well for small test sets (Zheng and Webb 2000).

**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 Naïve Bayes for classification problems and linear regression for regression problems (Section 6.5, pages 251–253). 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.

### Miscellaneous classifiers

The misc. category includes two simple classifiers that were mentioned at the end of Section 4.7 (page 136). **Hyperpipes**, for discrete classification problems, records the range of values observed in the training data for each attribute and category and works out which ranges contain the attribute values of a test instance, choosing the category with the largest number of correct ranges. **VFI** (voting feature intervals) constructs intervals around each class by discretizing numeric attributes and using point intervals for nominal ones, records class counts for each interval on each attribute, and classifies test instances by voting (Demiroz and Guvenir 1997). A simple attribute weighting scheme assigns higher weight to more confident intervals, where confidence is a function of entropy. **VFI** is faster than Naïve Bayes but slower than **hyperpipes**. Neither method can handle missing values.

### 10.5 Metalearning algorithms

Metalearning algorithms, listed in Table 10.6, take classifiers and turn them into more powerful learners. One parameter specifies the base classifier; others specify the number of iterations for schemes such as bagging and boosting and an initial seed for the random number generator. We already met **FilteredClassifier** in Section 10.3: 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.

**Bagging and randomization**

*Bagging* bags a classifier to reduce variance (Section 7.5, page 316). 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 (Section 7.5, page 320). This only makes sense if the base classifier is randomized; otherwise, all classifiers would be the same.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Meta</strong></td>
<td></td>
</tr>
<tr>
<td>AdaBoostM1</td>
<td>Boost using the AdaBoostM1 method</td>
</tr>
<tr>
<td>AdditiveRegression</td>
<td>Enhance the performance of a regression method by iteratively fitting the residuals</td>
</tr>
<tr>
<td>AttributeSelectedClassifier</td>
<td>Reduce dimensionality of data by attribute selection</td>
</tr>
<tr>
<td>Bagging</td>
<td>Bag a classifier; works for regression too</td>
</tr>
<tr>
<td>ClassificationViaRegression</td>
<td>Perform classification using a regression method</td>
</tr>
<tr>
<td>CostSensitiveClassifier</td>
<td>Make its base classifier cost sensitive</td>
</tr>
<tr>
<td>CVParameterSelection</td>
<td>Perform parameter selection by cross-validation</td>
</tr>
<tr>
<td>Decorate</td>
<td>Build ensembles of classifiers by using specially constructed artificial training examples</td>
</tr>
<tr>
<td>FilteredClassifier</td>
<td>Run a classifier on filtered data</td>
</tr>
<tr>
<td>Grading</td>
<td>Metalearners whose inputs are base-level predictions that have been marked as correct or incorrect</td>
</tr>
<tr>
<td>LogitBoost</td>
<td>Perform additive logistic regression</td>
</tr>
<tr>
<td>MetaCost</td>
<td>Make a classifier cost-sensitive</td>
</tr>
<tr>
<td>MultiBoostAB</td>
<td>Combine boosting and bagging using the MultiBoosting method</td>
</tr>
<tr>
<td>MultiClassClassifier</td>
<td>Use a two-class classifier for multiclass datasets</td>
</tr>
<tr>
<td>MultiScheme</td>
<td>Use cross-validation to select a classifier from several candidates</td>
</tr>
<tr>
<td>OrdinalClassClassifier</td>
<td>Apply standard classification algorithms to problems with an ordinal class value</td>
</tr>
<tr>
<td>RacedIncrementalLogitBoost</td>
<td>Batch-based incremental learning by racing logit-boosted committees</td>
</tr>
<tr>
<td>RandomCommittee</td>
<td>Build an ensemble of randomizable base classifiers</td>
</tr>
<tr>
<td>RegressionByDiscretization</td>
<td>Discretize the class attribute and employ a classifier</td>
</tr>
<tr>
<td>Stacking</td>
<td>Combine several classifiers using the stacking method</td>
</tr>
<tr>
<td>StackingC</td>
<td>More efficient version of stacking</td>
</tr>
<tr>
<td>ThresholdSelector</td>
<td>Optimize the F-measure for a probabilistic classifier</td>
</tr>
<tr>
<td>Vote</td>
<td>Combine classifiers using average of probability estimates or numeric predictions</td>
</tr>
</tbody>
</table>

Table 10.6  Metalearning algorithms in Weka.
**Boosting**

*AdaBoostM1* implements the algorithm described in Section 7.5 (page 321; Figure 7.7). 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). *MultiBoostAB* combines boosting with a variant of bagging to prevent overfitting (Webb 2000).

Whereas boosting only applies to nominal classes, *AdditiveRegression* enhances the performance of a regression learner (Section 7.5, page 325). 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.

*Decorate* builds ensembles of diverse classifiers by using specially constructed artificial training examples. This technique is claimed to consistently improve on the base classifier and on the bagging and random forest metalearners (Melville and Mooney, 2005). It outperforms boosting on small training sets and rivals it on larger ones. One parameter is the number of artificial examples to use as a proportion of the training data. Another is the desired number of classifiers in the ensemble, although execution may terminate prematurely because the number of iterations can also be capped. Larger ensembles usually produce more accurate models but have greater training time and model complexity.

*LogitBoost* performs additive logistic regression (Section 7.5, page 327). 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. *RacedIncrementalLogitBoost* learns by racing LogitBoosted committees, and operates incrementally by processing the data in batches (pages 347–348), making it useful for large datasets (Frank et al. 2002). Each committee member is learned from a different batch. The batch size starts at a given minimum and repeatedly doubles until it reaches a preset maximum. Resampling is used if the base classifier cannot handle weighted instances (you can also force resampling anyway). Log-likelihood pruning can be used within each committee: this discards new committee members if they decrease the log-likelihood based on the validation data. You can determine how many instances to hold out for validation. The validation data is also used to determine which committee to retain when training terminates.

---

6 The random forest scheme was mentioned on page 407. It is really a metalearner, but Weka includes it among the decision tree methods because it is hardwired to a particular classifier, *RandomTree*. 
Combining classifiers

Vote provides a baseline method for combining classifiers by averaging their probability estimates (classification) or numeric predictions (regression). MultiScheme selects the best classifier from a set of candidates using cross-validation of percentage accuracy (classification) or mean-squared error (regression). The number of folds is a parameter. Performance on training data can be used instead.

Stacking combines classifiers using stacking (Section 7.5, page 332) for both classification and regression problems. You specify the base classifiers, the meta-learner, and the number of cross-validation folds. StackingC implements a more efficient variant for which the metalearner must be a numeric prediction scheme (Seewald 2002). In Grading, the inputs to the metalearner are base-level predictions that have been marked (i.e., “graded”) as correct or incorrect. For each base classifier, a metalearned is learned that predicts when the base classifier will err. Just as stacking may be viewed as a generalization of voting, grading generalizes selection by cross-validation (Seewald and Fürnkranz 2001).

Cost-sensitive learning

There are two metalearners for cost-sensitive learning (Section 5.7). The cost matrix can be supplied as a parameter or loaded from a file in the directory set by the onDemandDirectory property, named by the relation name and with the extension cost. CostSensitiveClassifier either reweights training instances according to the total cost assigned to each class (cost-sensitive learning, page 165) or predicts the class with the least expected misclassification cost rather than the most likely one (cost-sensitive classification, page 164). MetaCost generates a single cost-sensitive classifier from the base learner (Section 7.5, pages 319–320). This implementation uses all bagging iterations when reclassifying training data (Domingos 1999 reports a marginal improvement when using only those iterations containing each training instance to reclassify it). You can specify each bag’s size and the number of bagging iterations.

Optimizing performance

Three metalearners use the wrapper technique to optimize the base classifier’s performance. AttributeSelectedClassifier selects attributes, reducing the data’s dimensionality before passing it to the classifier (Section 7.1, page 290). You can choose the attribute evaluator and search method using the Select attributes panel described in Section 10.2. 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 $P 1 10 11$. The number of cross-validation folds can be specified.
The third metalearner, *ThresholdSelector*, optimizes the F-measure (Section 5.7) by selecting a probability threshold on the classifier’s output. Performance can be 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
3. Whichever value is least frequent
4. Whichever value is most frequent
5. The first class named *yes*, *pos(itive)*, or 1.

**Retargeting classifiers for different tasks**

Four metalearners adapt learners designed for one kind of task to another. *ClassificationViaRegression* performs classification using a regression method by binarizing the class and building a regression model for each value. *Regression-ByDiscretization* is a regression scheme that discretizes 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. *OrdinalClassClassifier* applies standard classification algorithms to ordinal-class problems (Frank and Hall 2001). *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 (Section 7.5, page 334)
4. Randomly selected error-correcting codes

Random code vectors are known to have good error-correcting properties: a parameter specifies the length of the code vector (in bits).

### 10.6 Clustering algorithms

Table 10.7 lists Weka’s clustering algorithms; the first two and *SimpleKMeans* are described in Section 6.6. For the *EM* implementation you can specify how many clusters to generate or the algorithm can decide using cross-validation—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 minimum allowable standard deviation for the normal density calculation.
SimpleKMeans clusters data using \( k \)-means; the number of clusters is specified by a parameter. Cobweb implements both the Cobweb algorithm for nominal attributes and the Classit algorithm for numeric attributes. The ordering and priority of the merging and splitting operators differs between 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.

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 meta-clusterer that wraps a clustering algorithm to make it return a probability distribution and density. To each cluster it fits a discrete distribution or a symmetric normal distribution (whose minimum standard deviation is a parameter).

### Table 10.7 Clustering algorithms.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>( EM )</td>
<td>Cluster using expectation maximization</td>
</tr>
<tr>
<td>Cobweb</td>
<td>Implements the Cobweb and Classit clustering algorithms</td>
</tr>
<tr>
<td>FarthestFirst</td>
<td>Cluster using the farthest first traversal algorithm</td>
</tr>
<tr>
<td>MakeDensityBasedClusterer</td>
<td>Wrap a clusterer to make it return distribution and density</td>
</tr>
<tr>
<td>SimpleKMeans</td>
<td>Cluster using the ( k )-means method</td>
</tr>
</tbody>
</table>

### Table 10.8 Association-rule learners.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Apriori )</td>
<td>Find association rules using the Apriori algorithm</td>
</tr>
<tr>
<td>PredictiveApriori</td>
<td>Find association rules sorted by predictive accuracy</td>
</tr>
<tr>
<td>Tertius</td>
<td>Confirmation-guided discovery of association or classification rules</td>
</tr>
</tbody>
</table>

10.7 Association-rule learners

Weka has three association-rule learners, listed in Table 10.8. Apriori implements the Apriori algorithm (Section 4.5). 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 the examples covered by the premise that are also
covered by the consequent (called accuracy in Section 4.5); Lift, which is deter-
dined by dividing the confidence by the support (called coverage in Section 4.5); Leverage, which is the proportion of additional examples covered by both the
premise and the consequent beyond those expected if the premise and conse-
quently 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.

PredictiveApriori combines confidence and support into a single measure of
predictive accuracy (Scheffer 2001) and finds the best \( n \) association rules in order. Internally, the algorithm successively increases the support threshold, because
the value of predictive accuracy depends on it. Tertius finds rules according to
a confirmation measure (Flach and Lachiche 1999), seeking rules with multiple
conditions in the consequent, like Apriori, but differing in that these conditions
are OR’d together, not ANDed. It can be set to find rules that predict a single
condition or a predetermined attribute (i.e., classification rules). One parameter
determines whether negation is allowed in the antecedent, the consequent,
or both; others give the number of rules sought, minimum degree of confir-
mation, minimum coverage, maximum proportion of counterinstances, and
maximum rule size. Missing values can match any value, never match, or be sig-
nificant and possibly appear in rules.

10.8 Attribute selection

Figure 10.21 shows that part of Weka’s attribute selection panel where you
specify the attribute evaluator and search method; Table 10.9 and Table 10.10
list the choices. Attribute selection is normally done by searching the space of
attribute subsets, evaluating each one (Section 7.1). This is achieved by com-
bining one of the four attribute subset evaluators in Table 10.9 with one of the
seven search methods in Table 10.10. A potentially faster but less accurate
approach is to evaluate the attributes individually and sort them, discarding
attributes that fall below a chosen cutoff point. This is achieved by selecting one of the eight single-attribute evaluators in Table 10.9 and using the ranking method in Table 10.10. The Weka interface allows both possibilities by letting the user choose a selection method from Table 10.9 and a search method from Table 10.10, producing an error message if you select an inappropriate combi-

<table>
<thead>
<tr>
<th>Table 10.9 Attribute evaluation methods for attribute selection.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>-----------------------</td>
</tr>
<tr>
<td>Attribute subset evaluator</td>
</tr>
<tr>
<td>CfsSubsetEval</td>
</tr>
<tr>
<td>ClassifierSubsetEval</td>
</tr>
<tr>
<td>ConsistencySubsetEval</td>
</tr>
<tr>
<td>WrapperSubsetEval</td>
</tr>
<tr>
<td>Single-attribute evaluator</td>
</tr>
<tr>
<td>ChiSquaredAttributeEval</td>
</tr>
<tr>
<td>GainRatioAttributeEval</td>
</tr>
<tr>
<td>InfoGainAttributeEval</td>
</tr>
<tr>
<td>OneRAttributeEval</td>
</tr>
<tr>
<td>PrincipalComponents</td>
</tr>
<tr>
<td>ReliefFAttributeEval</td>
</tr>
<tr>
<td>SVMAttributeEval</td>
</tr>
<tr>
<td>SymmetricalUncertAttributeEval</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 10.10 Search methods for attribute selection.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>-----------------------</td>
</tr>
<tr>
<td>Search method</td>
</tr>
<tr>
<td>BestFirst</td>
</tr>
<tr>
<td>ExhaustiveSearch</td>
</tr>
<tr>
<td>GeneticSearch</td>
</tr>
<tr>
<td>GreedyStepwise</td>
</tr>
<tr>
<td>RaceSearch</td>
</tr>
<tr>
<td>RandomSearch</td>
</tr>
<tr>
<td>RankSearch</td>
</tr>
<tr>
<td>Ranking method</td>
</tr>
<tr>
<td>Ranker</td>
</tr>
</tbody>
</table>
nation. The status line refers you to the error log for the message (see the end of Section 10.1).

**Attribute subset evaluators**

Subset evaluators take a subset of attributes and return a numeric 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 have low intercorrelation (Section 7.1). 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. ConsistencySubsetEval evaluates attribute sets by the degree of consistency in class values when the training instances are projected onto the set. The consistency of any subset of attributes can never improve on that of the full set, so this evaluator is usually used in conjunction with a random or exhaustive search that seeks the smallest subset whose consistency is the same as that of the full attribute set.

Whereas the previously mentioned subset evaluators are filter methods of attribute selection (Section 7.1), the remainder are wrapper methods. ClassifierSubsetEval uses a classifier, specified in the object editor as a parameter, to evaluate sets of attributes on the training data or on a separate holdout set. WrapperSubsetEval also uses a classifier to evaluate attribute sets, but it employs cross-validation to estimate the accuracy of the learning scheme for each set.

**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). They can also be used in the RankSearch method. ReliefFAttributeEval is instance-based: it samples instances randomly and checks neighboring instances of the same and different classes (Section 7.1). 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 dis-
tribute 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. SymmetricalUncertaintyEvaluator evaluates an attribute \(A\) by measuring its symmetric uncertainty with respect to the class \(C\) (Section 7.1, page 291).

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.

SVMAttributeEval evaluates attributes using recursive feature elimination with a linear support vector machine (Section 7.1, page 291). Attributes are selected one by one based on the size of their coefficients, relearning after each one. To speed things up a fixed number (or proportion) of attributes can be removed at each stage. Indeed, a proportion can be used until a certain number of attributes remain, thereupon switching to the fixed-number method—rapidly eliminating many attributes and then considering each one more intensively. Various parameters are passed to the support vector machine: complexity, epsilon, tolerance, and the filtering method used.

Unlike other single-attribute evaluators, PrincipalComponents transforms the set of attributes. The new attributes are ranked in order of their eigenvalues (Section 7.3, page 306); optionally, a subset is selected by choosing sufficient eigenvectors to account for a given proportion of the variance (95% by default). You can also use it to transform the reduced data back to the original space.

**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 indices) 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.

*RaceSearch*, used with *ClassifierSubsetEval*, calculates the cross-validation error of competing attribute subsets using race search (Section 7.1). The four different searches described on page 295 are implemented: forward selection, backward elimination, schemata search, and rank racing. In the last case a separate attribute evaluator (which can also be specified) is used to generate an initial ranking. Using forward selection, it is also possible to generate a ranked list of attributes by continuing racing until all attributes have been selected: the ranking is set to the order in which they are added. As with *GreedyStepwise*, you can specify the number of attributes to retain or set a threshold below which attributes are discarded.

*GeneticSearch* uses a simple genetic algorithm (Goldberg 1989). Parameters include population size, number of generations, and probabilities of crossover and mutation. You can specify a list of attribute indices as the starting point, which becomes a member of the initial population. Progress reports can be generated every so many generations. *RandomSearch* randomly searches the space of attribute subsets. If an initial set is supplied, it searches for subsets that improve on (or equal) the starting point and have fewer (or the same number of) attributes. Otherwise, it starts from a random point and reports the best subset found. Placing all attributes in the initial set yields Liu and Setiono’s (1996) probabilistic feature selection algorithm. You can determine the fraction of the search space to explore. *ExhaustiveSearch* searches through the space of attribute subsets, starting from the empty set, and reports the best subset found. If an initial set is supplied, it searches backward from this starting point and reports the smallest subset with a better (or equal) evaluation.

*RankSearch* sorts attributes using a single-attribute evaluator and then ranks promising subsets using an attribute subset evaluator. The latter is specified in the top box of Figure 10.21, as usual; the attribute evaluator is specified as a property in *RankSearch*’s object editor. It starts by sorting the attributes with the single-attribute evaluator and then evaluates subsets of increasing size using the subset evaluator—the best attribute, the best attribute plus the next best one, and so on—reporting the best subset. This procedure has low computational complexity: the number of times both evaluators are called is linear in the number of attributes. Using a simple single-attribute evaluator (e.g., *GainRatioAttributeEval*), the selection procedure is very fast.

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 in the lower part of Table 10.9—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 cutoff 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.
With the Knowledge Flow interface, users select Weka components from a tool bar, place them on a layout canvas, and connect them into a directed graph that processes and analyzes data. It provides an alternative to the Explorer for those who like thinking in terms of how data flows through the system. It also allows the design and execution of configurations for streamed data processing, which the Explorer cannot do. You invoke the Knowledge Flow interface by selecting KnowledgeFlow from the choices at the bottom of the panel shown in Figure 10.3(a).

11.1 Getting started

Here is a step-by-step example that loads an ARFF file and performs a cross-validation using J4.8. We describe how to build up the final configuration shown in Figure 11.1. First create a source of data by clicking on the DataSources tab (rightmost entry in the bar at the top) and selecting ARFFLoader from the
toolbar. The mouse cursor changes to crosshairs to signal that you should next place the component. Do this by clicking anywhere on the canvas, whereupon a copy of the ARFF loader icon appears there. To connect it to an ARFF file, right-click it to bring up the pop-up menu shown in Figure 11.2(a). Click Configure to get the file browser in Figure 11.2(b), from which you select the desired ARFF file. The File Format pull-down menu allows you to choose a different type of data source—for example, spreadsheet files.

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

We will perform cross-validation on the J48 classifier. In the data flow model, we first connect the CrossValidationFoldMaker to create the folds on which the classifier will run, and then pass its output to an object representing J48. CrossValidationFoldMaker is on the Evaluation panel. Select it, place it on the canvas, and connect it to the class assigner by right-clicking the latter and selecting
dataset from the menu (which is similar to that in Figure 11.2(a)). Next select J48 from the Classifiers panel and place a J48 component on the canvas. There are so many different classifiers that you have to scroll along the toolbar to find it. Connect J48 to the cross-validation fold maker in the usual way, but make the connection twice by first choosing trainingSet and then choosing testSet from the pop-up menu for the cross-validation fold maker. The next step is to select a ClassifierPerformanceEvaluator from the Evaluation panel and connect J48 to it by selecting the batchClassifier entry from the pop-up menu for J48. Finally, from the Visualization toolbar we place a TextViewer component on the canvas. Connect the classifier performance evaluator to it by selecting the text entry from the pop-up menu for the performance evaluator.

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

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

11.2 The Knowledge Flow components

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

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Visualization</strong></td>
<td></td>
</tr>
<tr>
<td>DataVisualizer</td>
<td>Visualize data in a 2D scatter plot</td>
</tr>
<tr>
<td>ScatterPlotMatrix</td>
<td>Matrix of scatter plots</td>
</tr>
<tr>
<td>AttributeSummarizer</td>
<td>Set of histograms, one for each attribute</td>
</tr>
<tr>
<td>ModelPerformanceChart</td>
<td>Draw ROC and other threshold curves</td>
</tr>
<tr>
<td>TextViewer</td>
<td>Visualize data or models as text</td>
</tr>
<tr>
<td>GraphViewer</td>
<td>Visualize tree-based models</td>
</tr>
<tr>
<td>StripChart</td>
<td>Display a scrolling plot of data</td>
</tr>
<tr>
<td><strong>Evaluation</strong></td>
<td></td>
</tr>
<tr>
<td>TrainingSetMaker</td>
<td>Make a dataset into a training set</td>
</tr>
<tr>
<td>TestSetMaker</td>
<td>Make a dataset into a test set</td>
</tr>
<tr>
<td>CrossValidationFoldMaker</td>
<td>Split a dataset into folds</td>
</tr>
<tr>
<td>TrainTestSplitMaker</td>
<td>Split a dataset into training and test sets</td>
</tr>
<tr>
<td>ClassAssigner</td>
<td>Assign one of the attributes to be the class</td>
</tr>
<tr>
<td>ClassValuePicker</td>
<td>Choose a value for the positive class</td>
</tr>
<tr>
<td>ClassifierPerformanceEvaluator</td>
<td>Collect evaluation statistics for batch evaluation</td>
</tr>
<tr>
<td>IncrementalClassifierEvaluator</td>
<td>Collect evaluation statistics for incremental evaluation</td>
</tr>
<tr>
<td>ClustererPerformanceEvaluator</td>
<td>Collect evaluation statistics for clusters</td>
</tr>
<tr>
<td>PredictionAppender</td>
<td>Append a classifier’s predictions to a dataset</td>
</tr>
</tbody>
</table>
shown in Figure 10.16(a). AttributeSummarizer gives a matrix of histograms, one for each attribute, like that in the lower right-hand corner of Figure 10.3(b). ModelPerformanceChart draws ROC curves and other threshold curves. GraphViewer pops up a panel for visualizing tree-based models, as in Figure 10.6(a). As before, you can zoom, pan, and visualize the instance data at a node (if it has been saved by the learning algorithm).

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

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

### 11.3 Configuring and connecting the components

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

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

Connections from a filter component are enabled when it receives input from a data source, whereupon follow-on dataset or instance connections can be made. Instance connections cannot be made to supervised filters or to unsu-

![Figure 11.3 Operations on the Knowledge Flow components.](image-url)
supervised filters that cannot handle data incrementally (such as Discretize). To get a test or training set out of a filter, you need to put the appropriate kind in.

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

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

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

### 11.4 Incremental learning

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

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

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

Figure 11.4(a) shows a configuration that works incrementally. An instance connection is made from the loader to the updatable Naïve Bayes classifier. The classifier’s text output is taken to a viewer that gives a textual description
of the model. Also, an \texttt{incrementalClassifier} connection is made to the corresponding performance evaluator. This produces an output of type \textit{chart}, which is piped to a strip chart visualization component to generate a scrolling data plot.

Figure 11.4(b) shows the strip chart output. It plots both the accuracy and the root mean-squared probability error against time. As time passes, the whole plot (including the axes) moves leftward to make room for new data at the right.
When the vertical axis representing time 0 can move left no farther, it stops and the time origin starts to increase from 0 to keep pace with the data coming in at the right. Thus when the chart is full it shows a window of the most recent time units. The strip chart can be configured to alter the number of instances shown on the x axis.

This particular Knowledge Flow configuration can process input files of any size, even ones that do not fit into the computer’s main memory. However, it all depends on how the classifier operates internally. For example, although they are incremental, many instance-based learners store the entire dataset internally.
The Explorer and Knowledge Flow environments help you determine how well machine learning schemes perform on given datasets. But serious investigative work involves substantial experiments—typically running several learning schemes on different datasets, often with various parameter settings—and these interfaces are not really suitable for this. The Experimenter enables you to set up large-scale experiments, start them running, leave them, and come back when they have finished and analyze the performance statistics that have been collected. They automate the experimental process. The statistics can be stored in ARFF format, and can themselves be the subject of further data mining. You invoke this interface by selecting Experimenter from the choices at the bottom of the panel in Figure 10.3(a).

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

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

![Weka Experiment Environment](image)

(a)

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

---

**Figure 12.1** (continued)

### Running an experiment

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

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

### Analyzing the results

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

The Analyze panel is shown in Figure 12.2. To analyze the experiment that has just been performed, click the Experiment button at the right near the top; otherwise, supply a file that contains the results of another experiment. Then click Perform test (near the bottom on the left). The result of a statistical signifi-
cance test of the performance of the first learning scheme (J48) versus that of the other two (OneR and ZeroR) will be displayed in the large panel on the right.

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

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

### 12.2 Simple setup

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

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

### 12.3 Advanced setup

The Experimenter has an advanced mode. Click near the top of the panel shown in Figure 12.1(a) to obtain the more formidable version of the panel shown in Figure 12.3. This enlarges the options available for controlling the experiment—including, for example, the ability to generate learning curves. However, the
advanced mode is hard to use, and the simple version suffices for most purposes. For example, in advanced mode you can set up an iteration to test an algorithm with a succession of different parameter values, but the same effect can be achieved in simple mode by putting the algorithm into the list several times with different parameter values. Something you may need the advanced mode for is to set up distributed experiments, which we describe in Section 12.5.

12.4 The Analyze panel

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

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

The Row and Column fields determine the dimensions of the comparison matrix. Clicking Select brings up a list of all the features that have been measured in the experiment—in other words, the column labels of the spreadsheet in Figure 12.1(c). You can select which to use as the rows and columns of the matrix. (The selection does not appear in the Select box because more than one parameter can be chosen simultaneously.) Figure 12.4 shows which items are selected for the rows and columns of Figure 12.2. The two lists show the experimental parameters (the columns of the spreadsheet). Dataset is selected for the rows (and there is only one in this case, the Iris dataset), and Scheme, Scheme options, and Scheme_version_ID are selected for the column (the usual convention of shift-clicking selects multiple entries). All three can be seen in Figure 12.2—in fact, they are more easily legible in the key at the bottom.
Figure 12.4 Rows and columns of Figure 12.2: (a) row field, (b) column field, (c) result of swapping the row and column selections, and (d) substituting Run for Dataset as rows.
If the row and column selections were swapped and the Perform test button were pressed again, the matrix would be transposed, giving the result in Figure 12.4(c). There are now three rows, one for each algorithm, and one column, for the single dataset. If instead the row of Dataset were replaced by Run and the test were performed again, the result would be as in Figure 12.4(d). Run refers to the runs of the cross-validation, of which there are 10, so there are now 10 rows. The number in parentheses after each row label (100 in Figure 12.4(c) and 10 in Figure 12.4(d)) is the number of results corresponding to that row—in other words, the number of measurements that participate in the averages displayed by the cells in that row. There is also a button that allows you to select a subset of columns to display (the baseline column is always included), and another that allows you to select the output format: plain text (default), output for the LaTeX typesetting system, and CSV format.

12.5 Distributing processing over several machines

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

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

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

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

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

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

To start the remote engine server, type

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

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

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

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

Start the Experimenter by typing

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

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

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

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

Distributing an experiment involves splitting it into subexperiments that RMI sends to the hosts for execution. By default, experiments are partitioned by dataset, in which case there can be no more hosts than there are datasets. Then each subexperiment is self-contained: it applies all schemes to a single dataset. An experiment with only a few datasets can be partitioned by run instead. For example, a 10 times 10-fold cross-validation would be split into 10 subexperiments, 1 per run.
Lurking behind Weka’s interactive interfaces—the Explorer, the Knowledge Flow, and the Experimenter—lies its basic functionality. This can be accessed in raw form through a command-line interface. Select Simple CLI from the interface choices at the bottom of Figure 10.3(a) to bring up a plain textual panel with a line at the bottom on which you enter commands. Alternatively, use the operating system’s command-line interface to run the classes in weka.jar directly, in which case you must first set the CLASSPATH environment variable as explained in Weka’s README file.

### 13.1 Getting started

At the beginning of Section 10.1 we used the Explorer to invoke the J4.8 learner on the weather data. To do the same thing in the command-line interface, type

```
java weka.classifiers.trees.J48 -t data/weather.arff
```
into the line at the bottom of the text panel. This incantation calls the Java virtual machine (in the Simple CLI, Java is already loaded) and instructs it to execute J4.8. Weka is organized in packages that correspond to a directory hierarchy. The program to be executed is called J48 and resides in the trees package, which is a subpackage of classifiers, which is part of the overall weka package. The next section gives more details of the package structure. The -t option signals that the next argument is the name of the training file: we are assuming that the weather data resides in a data subdirectory of the directory from which you fired up Weka. The result resembles the text shown in Figure 10.5. In the Simple CLI it appears in the panel above the line where you typed the command.

### 13.2 The structure of Weka

We have explained how to invoke filtering and learning schemes with the Explorer and connect them together with the Knowledge Flow interface. To go further, it is necessary to learn something about how Weka is put together. Detailed, up-to-date information can be found in the online documentation included in the distribution. This is more technical than the descriptions of the learning and filtering schemes given by the More button in the Explorer and Knowledge Flow’s object editors. It is generated directly from comments in the source code using Sun’s Javadoc utility. To understand its structure, you need to know how Java programs are organized.

**Classes, instances, and packages**

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

In Weka, the implementation of a particular learning algorithm is encapsulated in a class. For example, the J48 class described previously builds a C4.5 decision tree. Each time the Java virtual machine executes J48, it creates an instance of this class by allocating memory for building and storing a decision tree classifier. The algorithm, the classifier it builds, and a procedure for outputting the classifier are all part of that instantiation of the J48 class.
Larger programs are usually split into more than one class. The J48 class, for example, does not actually contain any code for building a decision tree. It includes references to instances of other classes that do most of the work. When there are a lot of classes—as in Weka—they become difficult to comprehend and navigate. Java allows classes to be organized into packages. A package is just a directory containing a collection of related classes: for example, the trees package mentioned previously contains the classes that implement decision trees. Packages are organized in a hierarchy that corresponds to the directory hierarchy: trees is a subpackage of the classifiers package, which is itself a subpackage of the overall weka package.

When you consult the online documentation generated by Javadoc from your Web browser, the first thing you see is an alphabetical list of all the packages in Weka, as shown in Figure 13.1(a). Here we introduce a few of them in order of importance.

The weka.core package

The core package is central to the Weka system, and its classes are accessed from almost every other class. You can determine what they are by clicking on the weka.core hyperlink, which brings up the Web page shown in Figure 13.1(b).

The Web page in Figure 13.1(b) is divided into two parts: the interface summary and the class summary. The latter is a list of classes contained within the package, and the former lists the interfaces it provides. An interface is similar to a class, the only difference being that it doesn’t actually do anything by itself—it is merely a list of methods without actual implementations. Other classes can declare that they “implement” a particular interface and then provide code for its methods. For example, the OptionHandler interface defines those methods that are implemented by all classes that can process command-line options, including all classifiers.

The key classes in the core package are Attribute, Instance, and Instances. An object of class Attribute represents an attribute. It contains the attribute’s name, its type, and, in the case of a nominal or string attribute, its possible values. An object of class Instance contains the attribute values of a particular instance; and an object of class Instances holds an ordered set of instances, in other words, a dataset. You can learn more about these classes by clicking their hyperlinks; we return to them in Chapter 14 when we show you how to invoke machine learning schemes from other Java code. However, you can use Weka from the command line without knowing the details.

Clicking the Overview hyperlink in the upper left corner of any documentation page returns you to the listing of all the packages in Weka that is shown in Figure 13.1(a).
### Packages
- weka.associations
- weka.associations.tertius
- weka.attributeSelection
- weka.classifiers
- weka.classifiers.bayes
- weka.classifiers.bayes.net
- weka.classifiers.bayes.net.estimate
- weka.classifiers.bayes.net.search
- weka.classifiers.bayes.net.search.fixed
- weka.classifiers.bayes.net.search.global
- weka.classifiers.bayes.net.search.local
- weka.classifiers.evaluation
- weka.classifiers.functions
- weka.classifiers.functions.neural
- weka.classifiers.functions.pace
- weka.classifiers.functions.supportVector
- weka.classifiers.lazy
- weka.classifiers.lazy_instance
- weka.classifiers.meta
- weka.classifiers.misc
- weka.classifiers.rules
- weka.classifiers.rules.part
- weka.classifiers.trees
- weka.classifiers.trees.adtree
- weka.classifiers.trees.j48
- weka.classifiers.trees.lmt
- weka.classifiers.trees.m5
- weka.clusterers
- weka.core
- weka.core.converters
- weka.datagen
- weka.ensemble
- weka.eval
- weka.filters
- weka.filters.supervised.attribute
- weka.filters.supervised.instance
- weka.filters.unsupervised.attribute
- weka.filters.unsupervised.instance
- weka.gui
- weka.gui.beans
- weka.gui.boundaryvisualizer
- weka.gui.experiment
- weka.gui.explore
- weka.gui.graphvisualizer
- weka.gui.streams
- weka.gui.treevisualizer
- weka.gui.vvisualize

### Overview

<table>
<thead>
<tr>
<th>Package</th>
<th>Overview</th>
<th>Package</th>
<th>Overview</th>
<th>Class</th>
<th>Overview</th>
<th>Class</th>
<th>Overview</th>
<th>Class</th>
<th>Overview</th>
<th>Class</th>
<th>Overview</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>weka.core</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Interface Summary

- **AdditionalMeasuresProducer**
  Interface to something that can measure something other than those calculated by evaluation modules.

- **Copyable**
  Interface implemented by classes that can produce "shallow" copies of their objects.

- **Drawable**
  Interface to something that can be drawn as a graph.

- **Matchable**
  Interface to something that can be matched with tree matching algorithms.

- **OptionHandler**
  Interface to something that understands options.

- **Randomizable**
  Interface to something that has random behaviour that is able to be seeded with an integer.

- **Summarizable**
  Interface to something that provides a short textual summary (as opposed to toString() which is usually a fairly complete description) of itself.

- **WeightedInstancesHandler**
  Interface to something that makes use of the information provided by instance weights.

### Class Summary

- **Attribute**
  Class for handling an attribute.

- **AttributeStats**
  A utility class that contains summary information on all the values that appear as a dataset for a particular attribute.

- **BinarySparseInstance**
  Class for storing a binary-data-only instance as a sparse vector.

- **CheckOptionHandler**
  Simple command line checking of classes that implement OptionHandler.

- **ContingencyTable**
  Class implementing some statistical routines for contingency tables.

- **FastVector**
  Implements a fast vector class without synchronized methods.

- **Instance**
  Class for handling an instance.

- **Instances**
  Class for handling an ordered set of weighted instances.

- **Mark**
  Class for performing operations on a matrix of floating-point values.

- **Optimization**
  Implementation of Active-sets method with BFGS update to solve optimization problem with only bounds constraints in multi-dimensions.

- **Option**
  Class to store information about an option.

- **ProtectedProperties**
  Simple class that extends the Properties class so that the properties are not able to be modified.

- **Queue**
  Class representing a FIFO queue.

- **RandomVariables**
  Class implementing some simple random variables generator.

- **Range**
  Class representing a range of cardinal numbers.

- **SelectedTag**
  Represents a selected value from a finite set of values, where each value is a Tag (i.e.,

- **SerializedObject**
  Class for storing an object in serialized form in memory.

- **SingleIndex**
  Class representing a single cardinal number.

- **SparseInstance**
  Class for storing an instance as a sparse vector.

- **SpecialFunctions**
  Class implementing some mathematical functions.

- **Statistics**
  Class implementing some distributions, tests, etc.

- **Tag**
  A tag simply associates a numeric ID with a String description.

- **Utils**
  Class implementing some simple utility methods.

**Figure 13.1** Using Javadoc: (a) the front page and (b) the `weka.core` package.
The weka.classifiers package

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

To see an example, click on weka.classifiers.trees and then on DecisionStump, which is a class for building a simple one-level binary decision tree (with an extra branch for missing values). Its documentation page, shown in Figure 13.2, shows the fully qualified name of this class, weka.classifiers.trees.DecisionStump, near the top. You have to use this rather lengthy name whenever you build a decision stump from the command line. The class name is sited in a small tree structure showing the relevant part of the class hierarchy. As you can see, DecisionStump is a subclass of weka.classifiers.Classifier, which is itself a subclass of java.lang.Object. The Object class is the most general one in Java: all classes are automatically subclasses of it.

After some generic information about the class—brief documentation, its version, and its author—Figure 13.2 gives an index of the constructors and methods of this class. A constructor is a special kind of method that is called whenever an object of that class is created, usually initializing the variables that collectively define its state. The index of methods lists the name of each one, the type of parameters it takes, and a short description of its functionality. Beneath those indices, the Web page gives more details about the constructors and methods. We will return to these details later.

As you can see, DecisionStump overwrites the distributionForInstance() method from Classifier: the default implementation of classifyInstance() in Classifier then uses this method to produce its classifications. In addition, it contains the toString(), toSource(), and main() methods. The first returns a textual description of the classifier, used whenever it is printed on the screen. The second is used to obtain a source code representation of the learned classifier. The third is called when you ask for a decision stump from the command line, in other words, every time you enter a command beginning with

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

Class DecisionStump

java.lang.Object
   |---weka.classifiers.Classifier
      |---weka.classifiers.trees.DecisionStump

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

public class DecisionStump
   extends Classifier
   implements WeightedInstancesHandler, Sourcable

Class for building and using a decision stump. Usually used in conjunction with a
boosting algorithm. Typical usage:

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

Version:
   SRevision: 1.17 $

Author:
   Eibe Frank (eibe@cs.waikato.ac.nz)

See Also:
   Serialized Form

Constructor Summary

DecisionStump ()

Figure 13.2 DecisionStump: A class of the weka.classifiers.trees package.
The presence of a `main()` method in a class indicates that it can be run from the command line and that all learning methods and filter algorithms implement it.

**Other packages**

Several other packages listed in Figure 13.1(a) are worth mentioning: `weka.associations`, `weka.clusterers`, `weka.estimators`, `weka.filters`, and `weka.attributeSelection`. The `weka.associations` package contains association rule learners. These have been placed in a separate package because association rules are fundamentally different from classifiers. The `weka.clusterers` package contains methods for unsupervised learning. The `weka.estimators` package contains subclasses of a generic `Estimator` class, which computes different types of probability distribution. These subclasses are used by the Naïve Bayes algorithm (among others).

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

As mentioned previously, all classes are automatically subclasses of `Object`. To examine the tree that corresponds to Weka’s hierarchy of classes, select the `Overview` link from the top of any page of the online documentation. Click `Tree` to display the overview as a tree that shows which classes are subclasses or superclasses of a particular class—for example, which classes inherit from `Classifier`.

The online documentation contains an index of all publicly accessible variables (called `fields`) and methods in Weka—in other words, all fields and methods that you can access from your own Java code. To view it, click `Overview` and then `Index`.

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

**13.3 Command-line options**

In the preceding example, the `-t` option was used in the command line to communicate the name of the training file to the learning algorithm. There are many other options that can be used with any learning scheme, and also scheme-specific ones that apply only to particular schemes. If you invoke a scheme without any command-line options at all, it displays the applicable options: first the general options, then the scheme-specific ones. In the command-line interface, type:

```
java weka.classifiers.trees.J48
```

You’ll see a list of the options common to all learning schemes, shown in Table 13.1, followed by those that apply only to `J48`, shown in Table 13.2. We will explain the generic options and then briefly review the scheme-specific ones.

**Generic options**

The options in Table 13.1 determine which data is used for training and testing, how the classifier is evaluated, and what kind of statistics are displayed. For example, the `-T` option is used to provide the name of the test file when evalu-
ating a learning scheme on an independent test set. By default the class is the last attribute in an ARFF file, but you can declare another one to be the class using \(-c\) followed by the position of the desired attribute, 1 for the first, 2 for the second, and so on. When cross-validation is performed (the default if a test file is not provided), the data is randomly shuffled first. To repeat the cross-validation several times, each time reshuffling the data in a different way, set the random number seed with \(-s\) (default value 1). With a large dataset you may want to reduce the number of folds for the cross-validation from the default value of 10 using \(-x\).

In the Explorer, cost-sensitive evaluation is invoked as described in Section 10.1. To achieve the same effect from the command line, use the \(-m\) option to provide the name of a file containing the cost matrix. Here is a cost matrix for the weather data:

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

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

It is also possible to save and load models. If you provide the name of an output file using \(-d\), Weka saves the classifier generated from the training data.

<table>
<thead>
<tr>
<th>Option</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-t) &lt;training file&gt;</td>
<td>Specify training file</td>
</tr>
<tr>
<td>(-T) &lt;test file&gt;</td>
<td>Specify test file; if none, a cross-validation is performed on the training data</td>
</tr>
<tr>
<td>(-c) &lt;class index&gt;</td>
<td>Specify index of class attribute</td>
</tr>
<tr>
<td>(-s) &lt;random number seed&gt;</td>
<td>Specify random number seed for cross-validation</td>
</tr>
<tr>
<td>(-x) &lt;number of folds&gt;</td>
<td>Specify number of folds for cross-validation</td>
</tr>
<tr>
<td>(-m) &lt;cost matrix file&gt;</td>
<td>Specify file containing cost matrix</td>
</tr>
<tr>
<td>(-d) &lt;output file&gt;</td>
<td>Specify output file for model</td>
</tr>
<tr>
<td>(-l) &lt;input file&gt;</td>
<td>Specify input file for model</td>
</tr>
<tr>
<td>(-o)</td>
<td>Output statistics only, not the classifier</td>
</tr>
<tr>
<td>(-i)</td>
<td>Output information retrieval statistics for two-class problems</td>
</tr>
<tr>
<td>(-k)</td>
<td>Output information-theoretical statistics</td>
</tr>
<tr>
<td>(-p) &lt;attribute range&gt;</td>
<td>Output predictions for test instances</td>
</tr>
<tr>
<td>(-v)</td>
<td>Output no statistics for training data</td>
</tr>
<tr>
<td>(-r)</td>
<td>Output cumulative margin distribution</td>
</tr>
<tr>
<td>(-z) &lt;class name&gt;</td>
<td>Output source representation of classifier</td>
</tr>
<tr>
<td>(-g)</td>
<td>Output graph representation of classifier</td>
</tr>
</tbody>
</table>
To evaluate the same classifier on a new batch of test data, you load it back using 
-l instead of rebuilding it. If the classifier can be updated incrementally, you 
can provide both a training file and an input file, and Weka will load the clas-
sifier and update it with the given training instances.

If you wish only to assess the performance of a learning scheme, use -o to 
suppress output of the model. Use -i to see the performance measures of pre-
cision, recall, and F-measure (Section 5.7). Use -k to compute information-
thoretical measures from the probabilities derived by a learning scheme 
(Section 5.6).

Weka users often want to know which class values the learning scheme actu-
ally predicts for each test instance. The -p option prints each test instance’s 
number, its class, the confidence of the scheme’s prediction, and the predicted 
class value. It also outputs attribute values for each instance and must be fol-
lowed by a specification of the range (e.g., 1–2)—use 0 if you don’t want any 
attribute values. You can also output the cumulative margin distribution for the 
training data, which shows how the distribution of the margin measure (Section 
7.5, page 324) changes with the number of boosting iterations. Finally, you can 
output the classifier’s source representation, and a graphical representation if 
the classifier can produce one.

**Scheme-specific options**

Table 13.2 shows the options specific to J4.8. You can force the algorithm to use 
the unpruned tree instead of the pruned one. You can suppress subtree raising, 
which increases efficiency. You can set the confidence threshold for pruning and 
the minimum number of instances permissible at any leaf—both parameters 
were described in Section 6.1 (page 199). As well as C4.5’s standard pruning

<table>
<thead>
<tr>
<th>Option</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>-U</td>
<td>Use unpruned tree</td>
</tr>
<tr>
<td>-C &lt;conf&gt;</td>
<td>Specify confidence threshold for pruning</td>
</tr>
<tr>
<td>-M &lt;num&gt;</td>
<td>Specify minimum number of instances in any leaf</td>
</tr>
<tr>
<td>-R</td>
<td>Use reduced-error pruning</td>
</tr>
<tr>
<td>-N &lt;fold&gt;</td>
<td>Specify number of folds for reduced-error pruning; use one fold as</td>
</tr>
<tr>
<td></td>
<td>pruning set</td>
</tr>
<tr>
<td>-B</td>
<td>Use binary splits only</td>
</tr>
<tr>
<td>-S</td>
<td>Don’t perform subtree raising</td>
</tr>
<tr>
<td>-L</td>
<td>Retain instance information</td>
</tr>
<tr>
<td>-A</td>
<td>Smooth the probability estimates using Laplace smoothing</td>
</tr>
<tr>
<td>-Q</td>
<td>Seed for shuffling data</td>
</tr>
</tbody>
</table>
procedure, reduced-error pruning (Section 6.2, pages 202–203) can be performed. The \(-N\) option governs the size of the holdout set: the dataset is divided equally into that number of parts and the last is held out (default value 3). You can smooth the probability estimates using the Laplace technique, set the random number seed for shuffling the data when selecting a pruning set, and store the instance information for future visualization. Finally, to build a binary tree instead of one with multiway branches for nominal attributes, use \(-B\).
When invoking learning schemes from the graphical user interfaces or the command line, there is no need to know anything about programming in Java. In this section we show how to access these algorithms from your own code. In doing so, the advantages of using an object-oriented programming language will become clear. From now on, we assume that you have at least some rudimentary knowledge of Java. In most practical applications of data mining the learning component is an integrated part of a far larger software environment. If the environment is written in Java, you can use Weka to solve the learning problem without writing any machine learning code yourself.

14.1 A simple data mining application

We present a simple data mining application for learning a model that classifies text files into two categories, *hit* and *miss*. The application works for arbitrary documents: We refer to them as *messages*. The implementation uses the
StringToWordVector filter mentioned in Section 10.3 (page 399) to convert the messages into attribute vectors in the manner described in Section 7.3. We assume that the program is called every time a new file is to be processed. If the Weka user provides a class label for the file, the system uses it for training; if not, it classifies it. The decision tree classifier J48 is used to do the work.

14.2 Going through the code

Figure 14.1 shows the source code for the application program, implemented in a class called MessageClassifier. The command-line arguments that the main() method accepts are the name of a text file (given by -m), the name of a file holding an object of class MessageClassifier (-t), and, optionally, the classification of the message in the file (-c). If the user provides a classification, the message will be converted into an example for training; if not, the MessageClassifier object will be used to classify it as hit or miss.

main()

The main() method reads the message into a Java StringBuffer and checks whether the user has provided a classification for it. Then it reads a MessageClassifier object from the file given by -t and creates a new object of class MessageClassifier if this file does not exist. In either case the resulting object is called messageCl. After checking for illegal command-line options, the program calls the method updateData() to update the training data stored in messageCl if a classification has been provided; otherwise, it calls classifyMessage() to classify it. Finally, the messageCl object is saved back into the file, because it may have changed. In the following sections, we first describe how a new MessageClassifier object is created by the constructor MessageClassifier() and then explain how the two methods updateData() and classifyMessage() work.

MessageClassifier()

Each time a new MessageClassifier is created, objects for holding the filter and classifier are generated automatically. The only nontrivial part of the process is creating a dataset, which is done by the constructor MessageClassifier(). First the dataset’s name is stored as a string. Then an Attribute object is created for each attribute, one to hold the string corresponding to a text message and the other for its class. These objects are stored in a dynamic array of type FastVector. (FastVector is Weka’s own implementation of the standard Java Vector class and is used throughout Weka for historical reasons.)

Attributes are created by invoking one of the constructors in the class Attribute. This class has a constructor that takes one parameter—the attribute’s name—and creates a numeric attribute. However, the constructor we use here
/**
 * Java program for classifying text messages into two classes.
 */

import weka.core.Attribute;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.FastVector;
import weka.core.Utils;
import weka.classifiers.Classifier;
import weka.classifiers.trees.J48;
import weka.filters.Filter;
import weka.filters.unsupervised.attribute.StringToWordVector;
import java.io.*;

public class MessageClassifier implements Serializable {

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

    /* The filter used to generate the word counts. */
    private StringToWordVector m_Filter = new StringToWordVector();

    /* The actual classifier. */
    private Classifier m_Classifier = new J48();

    /* Whether the model is up to date. */
    private boolean m_UpToDate;

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

        String nameOfDataset = "MessageClassificationProblem";

        // Create vector of attributes.
        FastVector attributes = new FastVector(2);

        // Add attribute for holding messages.
        attributes.addElement(new Attribute("Message", (FastVector)null));

        // Figure 14.1 Source code for the message classifier.
    }
}
// Add class attribute.
FastVector classValues = new FastVector(2);
classValues.addElement("miss");
classValues.addElement("hit");
attributes.addElement(new Attribute("Class", classValues));

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

/** *
* Updates data using the given training message.
*/
public void updateData(String message, String classValue) throws Exception {
    // Make message into instance.
    Instance instance = makeInstance(message, m_Data);

    // Set class value for instance.
    instance.setClassValue(classValue);

    // Add instance to training data.
    m_Data.add(instance);
    m_UpToDate = false;
}

/** *
* Classifies a given message.
*/
public void classifyMessage(String message) throws Exception {
    // Check whether classifier has been built.
    if (m_Data.numInstances() == 0) {
        throw new Exception("No classifier available.");
    }

    // Check whether classifier and filter are up to date.
    if (!m_UpToDate) {

Figure 14.1 (continued)
// Initialize filter and tell it about the input format.
    m_Filter.setInputFormat(m_Data);

// Generate word counts from the training data.
    Instances filteredData = Filter.useFilter(m_Data, m_Filter);

// Rebuild classifier.
    m_Classifier.buildClassifier(filteredData);
    m_UpToDate = true;
}

// Make separate little test set so that message
// does not get added to string attribute in m_Data.
    Instances testset = m_Data.stringFreeStructure();

// Make message into test instance.
    Instance instance = makeInstance(message, testset);

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

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

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

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

    // Create instance of length two.
    Instance instance = new Instance(2);

    // Set value for message attribute
    Attribute messageAtt = data.attribute("Message");
    instance.setValue(messageAtt, messageAtt.addStringValue(text));

Figure 14.1 (continued)
// Give instance access to attribute information from the dataset.
instance.setDataset(data);
return instance;
}

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

try {

    // Read message file into string.
    String messageName = Utils.getOption('m', options);
    if (messageName.length() == 0) {
        throw new Exception("Must provide name of message file.");
    }
    FileReader m = new FileReader(messageName);
    StringBuffer message = new StringBuffer();
    int l;
    while ((l = m.read()) != -1) {
        message.append((char)l);
    }
    m.close();

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

    // If model file exists, read it, otherwise create new one.
    String modelName = Utils.getOption('o', options);
    if (modelName.length() == 0) {
        throw new Exception("Must provide name of model file.");
    }
    MessageClassifier messageCl;
    try {
        ObjectInputStream modelInObjectFile =
            new ObjectInputStream(new FileInputStream(modelName));
        messageCl = (MessageClassifier) modelInObjectFile.readObject();
        modelInObjectFile.close();
    } catch (FileNotFoundException e) {
        messageCl = new MessageClassifier();
    }
    }

Figure 14.1 (continued)
// Check if there are any options left
Utils.checkForRemainingOptions(options);

// Process message.
if (classValue.length() != 0) {
    messageCl.updateData(message.toString(), classValue);
} else {
    messageCl.classifyMessage(message.toString());
}

// Save message classifier object.
ObjectOutputStream modelOutObjectFile =
    new ObjectOutputStream(new FileOutputStream(modelName));
modelOutObjectFile.writeObject(messageCl);
modelOutObjectFile.close();
} catch (Exception e) {
    e.printStackTrace();
}
}

Figure 14.1 (continued)

takes two parameters: the attribute’s name and a reference to a FastVector. If this reference is null, as in the first application of this constructor in our program, Weka creates an attribute of type string. Otherwise, a nominal attribute is created. In that case it is assumed that the FastVector holds the attribute values as strings. This is how we create a class attribute with two values hit and miss: by passing the attribute’s name (class) and its values—stored in a FastVector—to Attribute().

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

Now that you know how to create an empty dataset, consider how the MessageClassifier object actually incorporates a new training message. The method `updateData()` does this job. It first converts the given message into a training instance by calling `makeInstance()`, which begins by creating an object of class `Instance` that corresponds to an instance with two attributes. The constructor of the `Instance` object sets all the instance’s values to be `missing` and its weight to 1. The next step in `makeInstance()` is to set the value of the string attribute holding the text of the message. This is done by applying the `setValue()` method of the `Instance` object, providing it with the attribute whose value needs to be changed, and a second parameter that corresponds to the new value’s index in the definition of the string attribute. This index is returned by the `addStringValue()` method, which adds the message text as a new value to the string attribute and returns the position of this new value in the definition of the string attribute.

Internally, an `Instance` stores all attribute values as double-precision floating-point numbers regardless of the type of the corresponding attribute. In the case of nominal and string attributes this is done by storing the index of the corresponding attribute value in the definition of the attribute. For example, the first value of a nominal attribute is represented by 0.0, the second by 1.0, and so on. The same method is used for string attributes: `addStringValue()` returns the index corresponding to the value that is added to the definition of the attribute.

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

Returning to `updateData()`, once the new instance has been returned from `makeInstance()` its class value is set and it is added to the training data. We also initialize `m_UpToDate`, a flag indicating that the training data has changed and the predictive model is hence not up to date.

classifyMessage()

Now let’s examine how MessageClassifier processes a message whose class label is unknown. The `classifyMessage()` method first checks whether a classifier has been built by determining whether any training instances are available. It then checks whether the classifier is up to date. If not (because the training data has changed) it must be rebuilt. However, before doing so the data must be converted into a format appropriate for learning using the `StringToWordVector` filter. First, we tell the filter the format of the input data by passing it a reference to the input dataset using `setInputFormat()`. Every time this method is called, the
filter is initialized—that is, all its internal settings are reset. In the next step, the data is transformed by useFilter(). This generic method from the Filter class applies a filter to a dataset. In this case, because StringToWordVector has just been initialized, it computes a dictionary from the training dataset and then uses it to form word vectors. After returning from useFilter(), all the filter’s internal settings are fixed until it is initialized by another call of inputFormat(). This makes it possible to filter a test instance without updating the filter’s internal settings (in this case, the dictionary).

Once the data has been filtered, the program rebuilds the classifier—in our case a J4.8 decision tree—by passing the training data to its buildClassifier() method and sets m_UpToDate to true. It is an important convention in Weka that the buildClassifier() method completely initializes the model’s internal settings before generating a new classifier. Hence we do not need to construct a new J48 object before we call buildClassifier().

Having ensured that the model stored in m_Classifier is current, we proceed to classify the message. Before makeInstance() is called to create an Instance object from it, a new Instances object is created to hold the new instance and passed as an argument to makeInstance(). This is done so that makeInstance() does not add the text of the message to the definition of the string attribute in m_Data. Otherwise, the size of the m_Data object would grow every time a new message was classified, which is clearly not desirable—it should only grow when training instances are added. Hence a temporary Instances object is created and discarded once the instance has been processed. This object is obtained using the method stringFreeStructure(), which returns a copy of m_Data with an empty string attribute. Only then is makeInstance() called to create the new instance.

The test instance must also be processed by the StringToWordVector filter before being classified. This is easy: the input() method enters the instance into the filter object, and the transformed instance is obtained by calling output(). Then a prediction is produced by passing the instance to the classifier’s classifyInstance() method. As you can see, the prediction is coded as a double value. This allows Weka’s evaluation module to treat models for categorical and numeric prediction similarly. In the case of categorical prediction, as in this example, the double variable holds the index of the predicted class value. To output the string corresponding to this class value, the program calls the value() method of the dataset’s class attribute.

There is at least one way in which our implementation could be improved. The classifier and the StringToWordVector filter could be combined using the FilteredClassifier metalearner described in Section 10.3 (page 401). This classifier would then be able to deal with string attributes directly, without explicitly calling the filter to transform the data. We didn’t do this because we wanted to demonstrate how filters can be used programmatically.
Suppose you need to implement a special-purpose learning algorithm that is not included in Weka. Or suppose you are engaged in machine learning research and want to investigate a new learning scheme. Or suppose you just want to learn more about the inner workings of an induction algorithm by actually programming it yourself. This section uses a simple example to show how to make full use of Weka’s class hierarchy when writing classifiers.

Weka includes the elementary learning schemes listed in Table 15.1, mainly for educational purposes. None take any scheme-specific command-line options. They are all useful for understanding the inner workings of a classifier. As an example, we describe the `weka.classifiers.trees.Id3` scheme, which implements the ID3 decision tree learner from Section 4.3.

### 15.1 An example classifier

Figure 15.1 gives the source code of `weka.classifiers.trees.Id3`, which, as you can see from the code, extends the `Classifier` class. Every classifier in Weka does so, whether it predicts a nominal class or a numeric one.
The first method in `weka.classifiers.trees.Id3` is `globalInfo()`: we mention it here before moving on to the more interesting parts. It simply returns a string that is displayed in Weka’s graphical user interfaces when this scheme is selected.

**buildClassifier()**

The `buildClassifier()` method constructs a classifier from a training dataset. In this case it first checks the data for a nonnominal class, missing attribute value, or any attribute that is not nominal, because the ID3 algorithm cannot handle these. It then makes a copy of the training set (to avoid changing the original data) and calls a method from `weka.core.Instances` to delete all instances with missing class values, because these instances are useless in the training process. Finally, it calls `makeTree()`, which actually builds the decision tree by recursively generating all subtrees attached to the root node.

**makeTree()**

The first step in `makeTree()` is to check whether the dataset is empty. If it is, a leaf is created by setting `m_Attribute` to null. The class value `m_ClassValue` assigned to this leaf is set to be missing, and the estimated probability for each of the dataset’s classes in `m_Distribution` is initialized to 0. If training instances are present, `makeTree()` finds the attribute that yields the greatest information gain for them. It first creates a Java enumeration of the dataset’s attributes. If the index of the class attribute is set—as it will be for this dataset—the class is automatically excluded from the enumeration.

Inside the enumeration, each attribute’s information gain is computed by `computeInfoGain()` and stored in an array. We will return to this method later. The `index()` method from `weka.core.Attribute` returns the attribute’s index in the dataset, which is used to index the array. Once the enumeration is complete, the attribute with the greatest information gain is stored in the instance variable `m_Attribute`. The `maxIndex()` method from `weka.core.Utils` returns the index of the greatest value in an array of integers or doubles. (If there is more than one element with the maximum value, the first is returned.) The index of this attribu-
Figure 15.1 Source code for the ID3 decision tree learner.
/**
 * Builds Id3 decision tree classifier.
 *
 * @param data the training data
 * @exception Exception if classifier can't be built successfully
 */
public void buildClassifier(Instances data) throws Exception {

    if (!data.classAttribute().isNominal()) {
        throw new UnsupportedClassTypeException("Id3: nominal class, please.");
    }

    Enumeration enumAtt = data.enumerateAttributes();
    while (enumAtt.hasMoreElements()) {
        if (!((Attribute) enumAtt.nextElement()).isNominal()) {
            throw new UnsupportedAttributeTypeException("Id3: only nominal " +
                "attributes, please.");
        }
    }

    Enumeration enum = data.enumerateInstances();
    while (enum.hasMoreElements()) {
        if (((Instance) enum.nextElement()).hasMissingValue()) {
            throw new NoSupportForMissingValuesException("Id3: no missing values, " +
                "please.");
        }
    }

    data = new Instances(data);
    data.deleteWithMissingClass();
    makeTree(data);
}

/**
 * Method for building an Id3 tree.
 *
 * @param data the training data
 * @exception Exception if decision tree can't be built successfully
 */
private void makeTree(Instances data) throws Exception {

    // Check if no instances have reached this node.
    if (data.numInstances() == 0) {
        m_Attribute = null;
    }

    // Code for building the decision tree...
}
m_ClassValue = Instance.missingValue();
m_Distribution = new double[data.numClasses()];
return;
}

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

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

/**
 * Classifies a given test instance using the decision tree.
 *
 * @param instance the instance to be classified
 */
```java
* @return the classification
*/
public double classifyInstance(Instance instance)
   throws NoSupportForMissingValuesException {

    if (instance.hasMissingValue()) {
      throw new NoSupportForMissingValuesException("Id3: no missing values, " + "please.");
    }
    if (m_Attribute == null) {
      return m_ClassValue;
    } else {
      return m_Successors[(int) instance.value(m_Attribute)].
          classifyInstance(instance);
    }
}

/**
* Computes class distribution for instance using decision tree.
*
* @param instance the instance for which distribution is to be computed
* @return the class distribution for the given instance
*/
public double[] distributionForInstance(Instance instance)
   throws NoSupportForMissingValuesException {

    if (instance.hasMissingValue()) {
      throw new NoSupportForMissingValuesException("Id3: no missing values, " + "please.");
    }
    if (m_Attribute == null) {
      return m_Distribution;
    } else {
      return m_Successors[(int) instance.value(m_Attribute)].
          distributionForInstance(instance);
    }
}

/**
* Prints the decision tree using the private toString method from below.
* 
Figure 15.1 (continued)
*/
public String toString() {
    if ((m_Distribution == null) && (m_Successors == null)) {
        return "Id3: No model built yet.";
    }
    return "Id3

    + toString(0);
}

/**
 * Computes information gain for an attribute.
 * 
 * @param data the data for which info gain is to be computed
 * @param att the attribute
 * @return the information gain for the given attribute and data
 */
private double computeInfoGain(Instances data, Attribute att)
    throws Exception {
    double infoGain = computeEntropy(data);
    Instances[] splitData = splitData(data, att);
    for (int j = 0; j < att.numValues(); j++) {
        if (splitData[j].numInstances() > 0) {
            infoGain -= ((double) splitData[j].numInstances() / 
                          (double) data.numInstances()) * 
                       computeEntropy(splitData[j]);
        }
    }
    return infoGain;
}

/**
 * Computes the entropy of a dataset.
 * 
 * @param data the data for which entropy is to be computed
 * @return the entropy of the data's class distribution
 */
private double computeEntropy(Instances data) throws Exception {
    double[] classCounts = new double[data.numClasses()];

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

/**
 * Splits a dataset according to the values of a nominal attribute.
 * @param data the data which is to be split
 * @param att the attribute to be used for splitting
 * @return the sets of instances produced by the split
 */
private Instances[] splitData(Instances data, Attribute att) {

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

/**
 * Outputs a tree at a certain level.

Figure 15.1 (continued)
* 
* @param level the level at which the tree is to be printed 
*/
private String toString(int level) {

StringBuilder text = new StringBuilder();

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

/**
 * Main method.
 * 
 * @param args the options for the classifier 
 */
public static void main(String[] args) {

try {
    System.out.println(Evaluation.evaluateModel(new Id3(), args));
} catch (Exception e) {
    System.err.println(e.getMessage());
}
}
ute is passed to the attribute() method from weka.core.Instances, which returns the corresponding attribute.

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

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

If an attribute with nonzero information gain is found, makeTree() splits the dataset according to the attribute’s values and recursively builds subtrees for each of the new datasets. To make the split it calls the method splitData(). This creates as many empty datasets as there are attribute values, stores them in an array (setting the initial capacity of each dataset to the number of instances in the original dataset), and then iterates through all instances in the original dataset, and allocates them to the new dataset that corresponds to the attribute’s value. It then reduces memory requirements by compacting the Instances objects. Returning to makeTree(), the resulting array of datasets is used for building subtrees. The method creates an array of Id3 objects, one for each attribute value, and calls makeTree() on each one by passing it the corresponding dataset.

computeInfoGain()

Returning to computeInfoGain(), the information gain associated with an attribute and a dataset is calculated using a straightforward implementation of the formula in Section 4.3 (page 102). First, the entropy of the dataset is computed. Then, splitData() is used to divide it into subsets, and computeEntropy() is called on each one. Finally, the difference between the former entropy and the weighted sum of the latter ones—the information gain—is returned. The method computeEntropy() uses the log2() method from weka.core.Utils to obtain the logarithm (to base 2) of a number.

classifyInstance()

Having seen how ID3 constructs a decision tree, we now examine how it uses the tree structure to predict class values and probabilities. Every classifier must implement the classifyInstance() method or the distributionForInstance() method (or both). The Classifier superclass contains default implementations
for both methods. The default implementation of classifyInstance() calls distributionForInstance(). If the class is nominal, it predicts the class with maximum probability, or a missing value if all probabilities returned by distributionForInstance() are zero. If the class is numeric, distributionForInstance() must return a single-element array that holds the numeric prediction, and this is what classifyInstance() extracts and returns. Conversely, the default implementation of distributionForInstance() wraps the prediction obtained from classifyInstance() into a single-element array. If the class is nominal, distributionForInstance() assigns a probability of one to the class predicted by classifyInstance() and a probability of zero to the others. If classifyInstance() returns a missing value, then all probabilities are set to zero. To give you a better feeling for just what these methods do, the weka.classifiers.trees.Id3 class overrides them both.

Let’s look first at classifyInstance(), which predicts a class value for a given instance. As mentioned in the previous section, nominal class values, like nominal attribute values, are coded and stored in double variables, representing the index of the value’s name in the attribute declaration. This is used in favor of a more elegant object-oriented approach to increase speed of execution. In the implementation of ID3, classifyInstance() first checks whether there are missing values in the instance to be classified; if so, it throws an exception. Otherwise, it descends the tree recursively, guided by the instance’s attribute values, until a leaf is reached. Then it returns the class value m_ClassValue stored at the leaf. Note that this might be a missing value, in which case the instance is left unclassified. The method distributionForInstance() works in exactly the same way, returning the probability distribution stored in m_Distribution.

Most machine learning models, and in particular decision trees, serve as a more or less comprehensible explanation of the structure found in the data. Accordingly, each of Weka’s classifiers, like many other Java objects, implements a toString() method that produces a textual representation of itself in the form of a String variable. ID3’s toString() method outputs a decision tree in roughly the same format as J4.8 (Figure 10.5). It recursively prints the tree structure into a String variable by accessing the attribute information stored at the nodes. To obtain each attribute’s name and values, it uses the name() and value() methods from weka.core.Attribute. Empty leaves without a class value are indicated by the string null.

main()

The only method in weka.classifiers.trees.Id3 that hasn’t been described is main(), which is called whenever the class is executed from the command line. As you can see, it’s simple: it basically just tells Weka’s Evaluation class to evaluate Id3 with the given command-line options and prints the resulting string. The one-line expression that does this is enclosed in a try–catch statement,
which catches the various exceptions that can be thrown by Weka’s routines or other Java methods.

The `evaluation()` method in `weka.classifiers.Evaluation` interprets the generic scheme-independent command-line options described in Section 13.3 and acts appropriately. For example, it takes the `-t` option, which gives the name of the training file, and loads the corresponding dataset. If there is no test file it performs a cross-validation by creating a classifier object and repeatedly calling `buildClassifier()` and `classifyInstance()` or `distributionForInstance()` on different subsets of the training data. Unless the user suppresses output of the model by setting the corresponding command-line option, it also calls the `toString()` method to output the model built from the full training dataset.

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

`OptionHandler` makes it possible to set options from the command line. To set them from within the graphical user interfaces, Weka uses the Java beans framework. All that is required are `set...()` and `get...()` methods for every parameter used by the class. For example, the methods `setPruningParameter()` and `getPruningParameter()` would be needed for a pruning parameter. There should also be a `pruningParameterTipText()` method that returns a description of the parameter for the graphical user interface. Again, see `weka.classifiers.rules.OneR` for an example.

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

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

In `weka.core` are many other useful interfaces for classifiers—for example, interfaces for classifiers that are `randomizable`, `summarizable`, `drawable`, and
15.2 Conventions for implementing classifiers

There are some conventions that you must obey when implementing classifiers in Weka. If you do not, things will go awry. For example, Weka’s evaluation module might not compute the classifier’s statistics properly when evaluating it.

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

Two other conventions have also been mentioned. One is that when a classifier can’t make a prediction, its `classifyInstance()` method must return `Instance.missingValue()` and its `distributionForInstance()` method must return probabilities of zero for all classes. The ID3 implementation in Figure 15.1 does this. Another convention is that with classifiers for numeric prediction, `classifyInstance()` returns the numeric value that the classifier predicts. Some classifiers, however, are able to predict nominal classes and their class probabilities, as well as numeric class values—`weka.classifiers.lazy.IBk` is an example. These implement the `distributionForInstance()` method, and if the class is numeric it returns an array of size 1 whose only element contains the predicted numeric value.

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


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


Scheffer, T. 2001. Finding association rules that trade support optimally against con-
fidence. In L. de Raedt and A. Siebes, editors, *Proceedings of the Fifth European
Conference on Principles of Data Mining and Knowledge Discovery*, Freiburg,


A new support vector regression algorithm. * Advances in Neural Information


Institute for Adaptive and Neural Computation, University of Edinburgh, UK.

Seewald, A. K. 2002. How to make stacking better and faster while also taking
care of an unknown weakness. *Proceedings of the Nineteenth International
Conference on Machine Learning*, Sydney, Australia. San Francisco: Morgan
Kaufmann, pp. 54–561.

Hoffmann, D. J. Hand, N. M. Adams, D. H. Fisher, and G. Guimarães, editors,
*Proceedings of the Fourth International Conference on Advances in Intelligent

Shafer, R., R. Agrawal, and M. Metha. 1996. SPRINT: A scalable parallel classifier for
data mining. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda,
editors, *Proceedings of the Second International Conference on Very Large
Databases*, Mumbai (Bombay), India. San Francisco: Morgan Kaufmann, pp.
544–555.

Cambridge, UK: Cambridge University Press.


ceptual dictionary. *Proceedings of the Fourteenth International Joint Conference
1314–1319


A
activation function, 234
acuity, 258
AdaBoost, 328
AdaBoost.M1, 321, 416
Add, 395
AddCluster, 396, 397
AddExpression, 397
additive logistic regression, 327–328
additive regression, 325–327
AdditiveRegression, 416
AddNoise, 400
AD (all-dimensions) tree, 280–283
ADTree, 408
advanced methods. See implementation—
real-world schemes
adversarial data mining, 356–358
aggregation, appropriate degree in data
warehousing, 53
Akaike Information Criterion (AIC), 277
Alberta Ingenuity Centre for Machine
Learning, 38
algorithms
additive logistic regression, 327
advanced methods, 187–283. See also
implementation—real-world schemes
association rule mining, 112–119
bagging, 319
basic methods, 83–142. See also algorithms-
basic methods
Bayesian network learning, 277–283
clustering, 136–139
clustering in Weka, 418–419
covering, 105–112
decision tree induction, 97–105
divide-and-conquer, 107
EM, 265–266
expanding examples into partial tree, 208
filtering in Weka, 393–403. See also filtering
algorithms
incremental, 346
instance-based learning, 128–136
learning in Weka, 403–414. See also learning
algorithms
linear models, 119–128
metalearning in Weka, 414–418
1R method, 84–88
perceptron learning rule, 124
RIPPER rule learner, 206
rule formation-incremental reduced-error
pruning, 205
separate-and-conquer, 112
statistical modeling, 88–97
stochastic, 348
Winnow, 127
See also individual subject headings.
alldimensions (AD) tree, 280–283
alternating decision tree, 329, 330, 343
Analyze panel, 443–445
analyzing purchasing patterns, 27
ancestor-of, 48
anomalies, 314–315
Index
anomaly detection systems, 357
antecedent, of rule, 65
AODE, 405
Apriori, 419
Apriori method, 141
area under the curve (AUC), 173
ARFF format, 53–55
    converting files to, 380–382
    Weka, 370, 371
ARFFLoader, 381, 427
arithmetic underflow, 276
assembling the data, 52–53
assessing performance of learning scheme, 286
assignment of key phrases, 353
Associate panel, 392
association learning, 43
association rules, 69–70, 112–119
    binary attributes, 119
    generating rules efficiently, 117–118
    item sets, 113, 114–115
    Weka, 419–420
association-rule learners in Weka, 419–420
attackers, 357
Attribute, 451
attribute(), 480
attribute discretization. See discretizing numeric attributes
attribute-efficient, 128
attribute evaluation methods in Weka, 421, 422–423
attribute filters in Weka, 394, 395–400, 402–403
attributeIndices, 382
attribute noise, 313
attribute-relation file format. See ARFF format
attributes, 49–52
    adding irrelevant, 288
    Boolean, 51
    class, 53
    as columns in tables, 49
    combinations of, 65
    continuous, 49
    discrete, 51
    enumerated, 51
    highly branching, 86
    identification code, 86
    independent, 267
    integer-valued, 49
    nominal, 49
    non-numeric, 17
    numeric, 49
    ordinal, 51
    relevant, 289
    rogue, 59
    selecting, 288
    subsets of values in, 80
    types in ARFF format, 56
    weighting, 237
See also orderings
AttributeSelectedClassifier, 417
attribute selection, 286–287, 288–296
    attribute evaluation methods in Weka, 421, 422–423
    backward elimination, 292, 294
    beam search, 293
    best-first search, 293
    forward selection, 292, 294
    race search, 295
    schemata search, 295
    scheme-independent selection, 290–292
    scheme-specific selection, 294–296
    searching the attribute space, 292–294
    search methods in Weka, 421, 423–425
    Weka, 392–393, 420–425
AttributeSelection, 403
attribute subset evaluators in Weka, 421, 422
AttributeSummarizer, 431
attribute transformations, 287, 305–311
    principal components analysis, 306–309
    random projections, 309
    text to attribute vectors, 309–311
    time series, 311
attribute weighting method, 237–238
AUC (area under the curve), 173
audit logs, 357
authorship ascription, 353
AutoClass, 269–270, 271
automatic data cleansing, 287, 312–315
    anomalies, 314–315
    improving decision trees, 312–313
    robust regression, 313–314
automatic filtering, 315
averaging over subnetworks, 283
axis-parallel class boundaries, 242

B
background knowledge, 348
backpropagation, 227–233
backtracking, 209
backward elimination, 292, 294
backward pruning, 34, 192
bagging, 316–319
Bagging, 414–415
bagging with costs, 319–320
bag of words, 95
balanced Winnow, 128
ball tree, 133–135
basic methods. See algorithms-basic methods
batch learning, 232
Bayes, Thomas, 141
Bayesian classifier. See Naïve Bayes
Bayesian clustering, 268–270
Bayesian multinet, 279–280
Bayesian network, 141, 271–283
  AD tree, 280–283
  Bayesian multinet, 279–280
  caveats, 276, 277
  counting, 280
  K2, 278
  learning, 276–283
  making predictions, 272–276
  Markov blanket, 278–279
  multiplication, 275
  Naïve Bayes classifier, 278
  network scoring, 277
  simplifying assumptions, 272
  structure learning by conditional independence tests, 280
  TAN (Tree Augmented Naïve Bayes), 279
  Weka, 403–406
Bayesian network learning algorithms, 277–283
Bayesian option trees, 328–331, 343
Bayesians, 141
Bayesian scoring metrics, 277–280, 283
Bayes information, 271
Bayes’s rule, 90, 181
beam search, 34, 293
beam width, 34
beer purchases, 27
Ben Ish Chai, 358
Bernoulli process, 147
BestFirst, 423
best-first search, 293
best-matching node, 257
bias, 32
  defined, 318
  language, 32–33
  multilayer perceptrons, 225, 226
  overfitting-avoidance, 34–35
  perceptron learning rule, 124
  search, 33–34
  what is it, 317
bias-variance decomposition, 317, 318
big data (massive datasets), 346–349
binning
  equal-frequency, 298
  equal-interval, 298
  equal-width, 342
binomial coefficient, 218
bits, 102
boolean, 51, 68
boosting, 321–325, 347
boosting in Weka, 416
bootstrap aggregating, 318
bootstrap estimation, 152–153
British Petroleum, 28
buildClassifier(), 453, 472, 482

C
C4.5, 105, 198–199
C5.0, 199
calm computing, 359, 362
capitalization conventions, 310
CAPPS (Computer Assisted Passenger Pre-Screening System), 357
CART (Classification And Regression Tree), 29, 38, 199, 253
categorical attributes, 49. See also nominal attributes
category utility, 260–262
causal relations, 350
CfsSubsetEval, 422
chain rule (probability theory), 275
character sets, 310
ChiSquaredAttributeEval, 423
chi-squared test, 302
circular ordering, 51
city-block metric, 129
ClassAssigner, 431
class attribute, 43
class distribution, 304
class hierarchy in Weka, 471–483
classification, 121
Classification And Regression Tree (CART), 29, 38, 199, 253
classification learning, 43
classification problems, 42
classification rules, 65–69, 200–214
  converting decision trees to, 198
criteria for choosing tests, 200–201
decision list, 11
different from association rules, 42
global optimization, 205–207
good (worthwhile) rules, 202–205
missing values, 201–202
numeric attributes, 202
partial decision tree, 207–210
pruning, 203, 205
replicated subtree problem, 66–68
RIPPER rule learner, 205
rules with exceptions, 210–213
Weka, 408–409
Classifier, 453
classifier in Weka, 366, 471–483
classifier algorithms. See learning algorithms
classifier errors in Weka, 379
ClassifierPerformanceEvaluator, 431
classifiers package, 453–455
ClassifierSubsetEval, 422
Classifier superclass, 480
classifyInstance(), 453, 480–481
Classify panel, 373, 384
classify text files into two categories, 461–469
Classit, 271
class noise, 313
ClassOrder, 403
class summary, 451
ClassValuePicker, 431
cleaning data, 52, 60
  automatic, 312
closed world assumption, 45
ClustererPerformanceEvaluator, 431
clustering, 43, 136–139, 254–271
  anomalies, 258
  basic assumption, 270
  Bayesian, 268–270
  category utility, 260–262
  Cluster panel (Weka), 391–392
document, 353
  EM algorithm, 265–266
  faster distance calculations, 138–139
  hierarchical, 139
  how many clusters?, 254–255
  incremental, 255–260
  k-means, 137–138
  MDL principle, 183–184
  merging, 257
  mixture model, 262–264, 266–268
  output, 81–82
  probability-based, 262–265
  RBF network, 234
  splitting, 254–255, 257
  unlabeled data, 337–339
  clustering algorithms in Weka, 418–419
  clustering for classification, 337
  ClusterMembership, 396, 397
  Cluster panel, 391–392
  Cobweb, 271
  Cobweb in Weka, 419
  co-EM, 340
  column separation, 336
  combining multiple models, 287, 315–336
    additive logistic regression, 327–328
    additive regression, 325–327
    bagging, 316–319
    bagging with costs, 319–320
    boosting, 321–325
    error-correcting output codes, 334–336
    logistic model trees, 331
    option trees, 328–331
randomization, 320–321
stacking, 332–334
command-line interface, 449–459
class, 450, 452
classifiers package, 453–455
core package, 451, 452
generic options, 456–458
instance, 450
Javadoc indices, 456
package, 451, 452
scheme-specific options, 458–459
starting up, 449–450
weka.associations, 455
weka.attributeSelection, 455
weka.clusterers, 455
weka.estimators, 455
weka.filters, 455
command-line options, 456–459
comma-separated value (CSV) format, 370, 371
comparing data mining methods, 153–157
ComplementNaiveBayes, 405
compression techniques, 362
computational learning theory, 324
calculateEntropy(), 480
Computer Assisted Passenger Pre-Screening System (CAPPS), 357
computer network security, 357
computer software. See Weka workbench
concept, 42
concept description, 42
concept description language, 32
concept representation, 82. See also knowledge representation
conditional independence, 275
conditional likelihood for scoring networks, 280, 283
confidence, 69, 113, 324
Confidence, 420
certainty tests, 154–157, 184
conflict resolution strategies, 82
colusionation matrix, 163
conjunction, 65
ConjunctiveRule, 408–409
consensus filter, 342
consequent, of rule, 65
ConsistencySubsetEval, 422
constrainted quadratic optimization, 217
consumer music, 359
contact lens data, 6, 13–15
continuous attributes, 49. See also numeric attributes
continuous monitoring, 28–29
converting discrete to numeric attributes, 304–305
convex hull, 171, 216
Conviction, 420
Copy, 395
core package, 451, 452
corrected resampled t-test, 157
correlation coefficient, 177–179
cost curves, 173–176
cost matrix, 164–165
cost of errors, 161–176
bagging, 319–320
cost curves, 173–176
cost-sensitive classification, 164–165
cost-sensitive learning, 165–166
Kappa statistic, 163–164
lift charts, 166–168
recall-precision curves, 171–172
ROC curves, 168–171
cost-sensitive classification, 164–165
CostSensitiveClassifier, 417
cost-sensitive learning, 165–166
cost-sensitive learning in Weka, 417
co-training, 339–340
covariance matrix, 267, 307
coverage, of association rules, 69
covering algorithm, 106–111
cow culling, 3–4, 37, 161–162
CPU performance data, 16–17
credit approval, 22–23
cross-validated ROC curves, 170
cross-validation, 149–152, 326
inner, 286
outer, 286
repeated, 144
CrossValidationFoldMaker, 428, 431
CSV format, 370, 371
CSVLoader, 381

cumulative margin distribution in Weka, 458

curves
  cost, 173
  lift, 166
  recall-precision, 171
  ROC, 168

customer support and service, 28
cutoff parameter, 260

CVParameterSelection, 417
cybersecurity, 29

D

dairy farmers (New Zealand), 3–4, 37, 161–162
data assembly, 52–53
data cleaning, 52–60. See also automatic data cleansing
data engineering. See engineering input and output
data integration, 52
data mining, 4–5, 9
data ownership rights, 35
data preparation, 52–60
data transformation. See attribute transformations

DataVisualizer, 389, 390, 430
data warehouse, 52–53
date attributes, 55
decision list, 11, 67
decision nodes, 328
decision stump, 325

DecisionStump, 407, 453, 454
decision table, 62, 295
DecisionTable, 408
decision tree, 14, 62–65, 97–105
  complexity of induction, 196
  converting to rules, 198
  data cleaning, 312–313
  error rates, 192–196
  highly branching attributes, 102–105
  missing values, 63, 191–192
  multiclass case, 107
  multivariate, 199
  nominal attribute, 62
  numeric attribute, 62, 189–191
  partial, 207–210
  pruning, 192–193, 312
  replicated subtree, 66
  rules, 198
  subtree raising, 193, 197
  subtree replacement, 192–193, 197
  three-way split, 63
  top-down induction, 97–105, 196–198
  two-way split, 62
  univariate, 199
  Weka, 406–408
  Weka’s User Classifier facility, 63–65

Decorate, 416
deduction, 350
default rule, 110
degrees of freedom, 93, 155
delta, 311
dendrograms, 82
denormalization, 47
density function, 93
diagnosis, 25–26
dichotomy, 51
directed acyclic graph, 272
direct marketing, 27
discrete attributes, 50. See also nominal attributes

Discretize, 396, 398, 402
discretizing numeric attributes, 287, 296–305
  chi-squared test, 302
  converting discrete to numeric attributes, 304–305
  entropy-based discretization, 298–302
  error-based discretization, 302–304
  global discretization, 297
  local discretization, 297
  supervised discretization, 297, 298
  unsupervised discretization, 297–298
  Weka, 398

disjunction, 32, 65
disjunctive normal form, 69
distance functions, 128–129, 239–242
distributed experiments in Weka, 445
distribution, 304
distributionForInstance(), 453, 481
divide-and-conquer. See decision tree
document classification, 94–96, 352–353
document clustering, 353
domain knowledge, 20, 33, 349–351
double-consequent rules, 118
duplicate data, 59
dynamic programming, 302

E
early stopping, 233
easy instances, 322
ecological applications, 23, 28
eigenvalue, 307
eigenvector, 307
Einstein, Albert, 180
electricity supply, 24–25
electromechanical diagnosis application, 144
11-point average recall, 172
EM, 418
EM algorithm, 265–266
EM and co-training, 340–341
EM procedure, 337–338
embedded machine learning, 461–469
engineering input and output, 285–343
  attribute selection, 288–296
  combining multiple models, 315–336
  data cleansing, 312–315
  discretizing numeric attributes, 296–305
  unlabeled data, 337–341
  See also individual subject headings
entity extraction, 353
entropy, 102
entropy-based discretization, 298–302
enumerated attributes, 50. See also nominal attributes
enumerating the concept space, 31–32
Epicurus, 183
epoch, 412
equal-frequency binning, 298
equal-interval binning, 298
equal-width binning, 342
erroneous values, 59
error-based discretization, 302–304
error-correcting output codes, 334–336
error log, 378
error rate
  bias, 317
  cost of errors. See cost of errors
decision tree, 192–196
defined, 144
training data, 145
“Essay towards solving a problem in the doctrine of chances, An” (Bayes), 141
ethics, 35–37
Euclidean distance, 78, 128, 129, 237
evaluation, 143–185
  bootstrap procedure, 152–153
  comparing data mining methods, 153–157
cost of errors, 161–176. See also cost of errors
cross-validation, 149–152
leave-one-out cross-validation, 151–152
MDL principle, 179–184
numeric prediction, 176–179
predicting performance, 146–149
predicting probabilities, 157–161
training and testing, 144–146
evaluation(), 482
evaluation components in Weka, 430, 431
Evaluation panel, 431
example problems
  contact lens data, 6, 13–15
  CPU performance data, 16–17
  iris dataset, 15–16
  labor negotiations data, 17–18, 19
  soybean data, 18–22
  weather problem, 10–12
exceptions, 70–73, 210–213
exclusive-or problem, 67
exemplar
  defined, 236
  generalized, 238–239
  noisy, 236–237
  redundant, 236
exemplar generalization, 238–239, 243
ExhaustiveSearch, 424
Expand all paths, 408
expectation, 265, 267
expected error, 174
expected success rate, 147
Experimenter, 437–447
  advanced panel, 443–445
  Analyze panel, 443–445
  analyzing the results, 440–441
  distributing processing over several machines, 445–447
  running an experiment, 439–440
  simple setup, 441–442
  starting up, 438–441
  subexperiments, 447
Explorer, 369–425
  ARFF, 370, 371, 380–382
  Associate panel, 392
  association-rule learners, 419–420
  attribute evaluation methods, 421, 422–423
  attribute selection, 392–393, 420–425
  boosting, 416
  classifier errors, 379
  Classify panel, 384
  clustering algorithms, 418–419
  Cluster panel, 391–392
  CSV format, 370, 371
  error log, 378
  file format converters, 380–382
  filtering algorithms, 393–403
  filters, 382–384
  J4.8, 373–377
  learning algorithms, 403–414. See also learning algorithms
  metalearning algorithms, 414–418
  models, 377–378
  panels, 380
  Preprocess panel, 380
  search methods, 421, 423–425
  Select attributes panel, 392–393
  starting up, 369–379
  supervised filters, 401–403
  training/testing learning schemes, 384–387
  unsupervised attribute filters, 395–400
  unsupervised instance filters, 400–401
  User Classifier, 388–391
  Visualize panel, 393
  extraction problems, 353, 354

F
  Fahrenheit, Daniel, 51
  fallback heuristic, 239
  false negative (FN), 162
  false positive (FP), 162
  false positive rate, 163
  False positive rate, 378
  Familiar, 360
  family tree, 45
    tabular representation of, 46
  FarthestFirst, 419
  features. See attributes
  feature selection, 341. See also attribute selection
  feedforward networks, 233
  fielded applications, 22
    continuous monitoring, 28–29
    customer support and service, 28
    cybersecurity, 29
    diagnosis, 25–26
    ecological applications, 23, 28
    electricity supply, 24–25
    hazard detection system, 23–24
    load forecasting, 24–25
    loan application, 22–23
    manufacturing processes, 28
    marketing and sales, 26–28
    oil slick detection, 23
    preventive maintenance of electromechanical devices, 25–26
    scientific applications, 28
  file format converters, 380–382
  file mining, 49
  filter, 290
  filter in Weka, 382–384
  FilteredClassifier, 401, 414
  filtering algorithms in Weka, 393–403
    sparse instances, 401
    supervised filters, 401–403
    unsupervised attribute filters, 395–400
    unsupervised instance filters, 400–401
  filtering approaches, 315
  filters menu, 383
  finite mixture, 262, 263
  FirstOrder, 399
Fisher, R. A., 15
flat file, 45
F-measure, 172
FN (false negatives), 162
folds, 150
forward pruning, 34, 192
forward selection, 292, 294
forward stagewise additive modeling, 325–327
Fourier analysis, 25
FP (false positives), 162
freedom, degrees of 93, 155
functional dependencies, 350
functions in Weka, 404–405, 409–410

G

gain ratio, 104
GainRatioAttributeEval, 423
gambling, 160
garbage in, garbage out. See cost of errors; data cleaning; error rate
Gaussian-distribution assumption, 92
Gaussian kernel function, 252
generalization as search, 30–35
bias, 32–35
enumerating concept space, 31–32
generalized distance functions, 241–242
generalized exemplars, 236
general-to-specific search bias, 34
genetic algorithms, 38
genetic algorithm search procedures, 294, 341
GeneticSearch, 424
getOptions(), 482
getting to know your data, 60
global discretization, 297
globalInfo(), 472
global optimization, 205–207
Gosset, William, 184
gradient descent, 227, 229, 230
Grading, 417
graphical models, 283
GraphViewer, 431
gray bar in margin of textbook (optional sections), 30
greedy search, 33

GreedyStepwise, 423–424
growing set, 202

H
Hamming distance, 335
hand-labeled data, 338
hapax legomena, 310
hard instances, 322
hash table, 280
hazard detection system, 23–24
hidden attributes, 272
hidden layer, 226, 231, 232
hidden units, 226, 231, 234
hierarchical clustering, 139
highly-branching attribute, 86
high-performance rule inducers, 188
histogram equalization, 298
historical literary mystery, 358
holdout method, 146, 149–150, 333
homeland defense, 357
HTML, 355
hypermetrope, 13
hyperpipes, 139
Hyperpipes, 414
hyperplane, 124, 125
hyperrectangle, 238–239
hyperspheres, 133
hypertext markup language (HTML), 355
hypothesis testing, 29

I
IB1, 413
IB3, 237
IBk, 413
ID3, 105
Id3, 404
identification code, 86, 102–104
implementation—real-world schemes, 187–283
Bayesian networks, 271–283
classification rules, 200–214
clustering, 254–271
decision tree, 189–199
instance-based, 236–243
linear models, 214–235
implementation—real-world schemes
(continued)
numeric prediction, 243–254
See also individual subject headings
inaccurate values, 59–60. See also cost of errors;
data cleaning; error rate
incremental algorithms, 346
incrementalClassifier, 434
IncrementalClassifierEvaluator, 431
incremental clustering, 255–260
incremental learning in Weka, 433–435
incremental reduced-error pruning, 203, 205
independent attributes, 267
index(), 472
induction, 29
inductive logic programming, 48, 60, 75, 351
Induct system, 214
industrial usage. See implementation—real-
world schemes
inferring rudimentary rules, 84–88
InfoGainAttributeEval, 422–423
informational loss function, 159–160, 161
information-based heuristic, 201
information extraction, 354
information gain, 99
information retrieval, 171
information value, 102
infrequent words, 353
inner cross-validation, 286
input, 41–60
ARFF format, 53–55
assembling the data, 52–53
attribute, 49–52
attribute types, 56–57
concept, 42–45
data engineering, 286–287, 288–315. See also
engineering input and output
data preparation, 52–60
getting to know your data, 60
inaccurate values, 59–60
instances, 45
missing values, 58
sparse data, 55–56
input layer, 224
instance in Weka, 450
Instance, 451
instance-based learning, 78, 128–136, 235–243
ball tree, 133–135
distance functions, 128–129, 239–242
finding nearest neighbors, 129–135
generalized distance functions, 241–242
generalized exemplars, 236
kD-trees, 130–132
missing values, 129
pruning noisy exemplars, 236–237
redundant exemplars, 236
simple method, 128–136, 235–236
weighting attributes, 237–238
Weka, 413–414
instance-based learning methods, 291
instance-based methods, 34
instance-based representation, 76–80
instance filters in Weka, 394, 400–401, 403
instances, 45
Instances, 451
instance space, 79
instance weights, 166, 321–322
integer-valued attributes, 49
intensive care patients, 29
interval, 88
interval quantities, 50–51
intrusion detection systems, 357
invertSelection, 382
in vitro fertilization, 3
iris dataset, 15–16
iris setosa, 15
iris versicolor, 15
iris virginica, 15
ISO-8601 combined date and time format, 55
item, 113
item sets, 113, 114–115
iterative distance-based clustering, 137–138
J
J4.8, 373–377
J48, 404, 450
Javadoc indices, 456
JDBC database, 445
JRip, 409
junk email filtering, 356–357
K
K2, 278
Kappa statistic, 163–164
kD-trees, 130–132, 136
Kepler’s three laws of planetary motion, 180
Kernel
  defined, 235
  perceptron, 223
  polynomial, 218
  RBF, 219
  sigmoid, 219
kernel density estimation, 97
kernel logistic regression, 223
kernel perceptron, 222–223
k-means, 137–138
k-nearest-neighbor method, 78
Knowledge Flow interface, 427–435
  configuring/connecting components, 431–433
  evaluation components, 430, 431
  incremental learning, 433–435
  starting up, 427
  visualization components, 430–431
knowledge representation, 61–82
  association rules, 69–70. See also association rules
  classification rules, 65–69. See also classification rules
  clusters, 81–82. See also clustering decision table, 62
decision tree, 62–65. See also decision tree instance-based representation, 76–80
rules involving relations, 73–75
rules with exceptions, 70–73, 210–213
trees for numeric prediction, 76
KStar, 413

lazy classifiers in Weka, 405, 413–414
LBR, 414
learning, 7–9
learning algorithms in Weka, 403–404
  algorithm, listed, 404–405
  Bayesian classifier, 403–406
  functions, 404–405, 409–410
  lazy classifiers, 405, 413–414
  miscellaneous classifiers, 405, 414
  neural network, 411–413
  rules, 404, 408–409
trees, 404, 406–408
learning rate, 229, 230
least-absolute-error regression, 220
LeastMedSq, 409–410
leave-one-out cross-validation, 151–152
levels of measurement, 50
level-0 model, 332
level-1 model, 332
Leverage, 420
Lift, 420
lift chart, 166–168, 172
lift factor, 166
linear classification, 121–128
linearly separable, 124
linear machine, 142
linear models, 119–128, 214–235
  backpropagation, 227–233
  computational complexity, 218
  kernel perceptron, 222–223
  linear classification, 121–128
  linear regression, 119–121
  logistic regression, 121–125
  maximum margin hyperplane, 215–217
  multilayer perceptrons, 223–226, 231, 233
  nonlinear class boundaries, 217–219
  numeric prediction, 119–120
  overfitting, 217–218
  perceptron, 124–126
  RBF network, 234
  support vector regression, 219–222
  Winnow, 126–128
linear regression, 77, 119–121
LinearRegression, 387, 409
linear threshold unit, 142

L
labor negotiations data, 17–18, 19
language bias, 32–33
language identification, 353
Laplace, Pierre, 91
Laplace estimator, 91, 267, 269
large datasets, 346–349
law of diminishing returns, 347

INDEX
listOptions(), 482
literary mystery, 358
LMT, 408
load forecasting, 24–25
loan application, 22–23
local discretization, 297
locally weighted linear regression, 244, 251–253, 253–254, 323
locally weighted Naïve Bayes, 252–253
Log button, 380
logic programs, 75
logistic model trees, 331
logistic regression, 121–125
LogitBoost, 328, 330, 331
LogitBoost, 416
logit transformation, 121
log-likelihood, 122–123, 276, 277
log-normal distribution, 268
log-odds distribution, 268
LWL, 414
M
M5’ program, 384
M5P, 408
M5Rules, 409
machine learning, 6
main(), 453
majority voting, 343
MakeDensityBasedClusterer, 419
MakeIndicator, 396, 398
makeTree(), 472, 480
Manhattan metric, 129
manufacturing processes, 28
margin, 324
margin curve, 324
market basket analysis, 27
market basket data, 55
marketing and sales, 26–28
Markov blanket, 278–279
Markov network, 283
massive datasets, 346–349
maximization, 265, 267
maximum margin hyperplane, 215–217
maxIndex(), 472
MDL metric, 277
MDL principle, 179–184
mean absolute error, 177–179
mean-squared error, 177, 178
measurement errors, 59
membership function, 121
memorization, 76
MergeTwoValues, 398
merging, 257
MetaCost, 319, 320
MetaCost, 417
metadata, 51, 349, 350
metadata extraction, 353
metalearner, 332
metalearning algorithms in Weka, 414–418
metric tree, 136
minimum description length (MDL) principle, 179–184
miscellaneous classifiers in Weka, 405, 414
missing values, 58
classification rules, 201–202
decision tree, 63, 191–192
instance-based learning, 129
1R, 86
mixture model, 267–268
model tree, 246–247
statistical modeling, 92–94
mixed-attribute problem, 11
mixture model, 262–264, 266–268
MLnet, 38
ModelPerformanceChart, 431
model tree, 76, 77, 243–251
building the tree, 245
missing values, 246–247
nominal attributes, 246
pruning, 245–246
pseudocode, 247–250
regression tree induction, compared, 243
replicated subtree problem, 250
rules, 250–251
smoothing, 244, 251
splitting, 245, 247
what is it, 250
momentum, 233
monitoring, continuous, 28–29
MultiBoostAB, 416
multiclass alternating decision trees, 329, 330, 343

*MultiClassClassifier*, 418

multiclass learning problems, 334

*MultilayerPerceptron*, 411–413

multilayer perceptrons, 223–226, 231, 233

multinomial distribution, 95

multinomial Naïve Bayes, 95, 96

multiple linear regression, 326

multiresponse linear regression, 121, 124

multistage decision property, 102

multivariate decision trees, 199

*MultiScheme*, 417

myope, 13

**N**

NaïveBayes, 403, 405

Naïve Bayes, 91, 278

  clustering for classification, 337–338
  co-training, 340
  document classification, 94–96
  limitations, 96–97
  locally weighted, 252–253
  multinomial, 95, 96
  power, 96
  scheme-specific attribute selection, 295–296
  selective, 296
  TAN (Tree Augmented Naïve Bayes), 279
  what can go wrong, 91

NaïveBayesMultinominal, 405

NaïveBayesSimple, 403

NaïveBayesUpdateable, 405

NBTree, 408

nearest-neighbor learning, 78–79, 128–136, 235, 242

nested exceptions, 213

nested generalized exemplars, 239

network scoring, 277

network security, 357

neural networks, 39, 233, 235, 253

neural networks in Weka, 411–413

n-gram profiles, 353, 361

Nnge, 409

noise

  data cleansing, 312

exemplars, 236–237

hand-labeled data, 338

robustness of learning algorithm, 306

noisy exemplars, 236–237

nominal attributes, 49, 236–237

  Cobweb, 271
  convert to numeric attributes, 304–305
  decision tree, 62
  mixture model, 267
  model tree, 246
  subset, 88

nominal quantities, 50

NominalToBinary, 398–399, 403

non-axis-parallel class boundaries, 242

Non-Bayesians, 141

nonlinear class boundaries, 217–219

NonSparseToSparse, 401

normal-distribution assumption, 92

normalization, 56

Normalize, 398, 400

normalize(), 480

normalized expected cost, 175

nuclear family, 47

null hypothesis, 155

numeric attribute, 49, 50, 56–57

  axis-parallel class boundaries, 242
  classification rules, 202
  Classit, 271
  converting discrete attributes to, 304–305
  decision tree, 62, 189–191
  discretizing, 296–305. *See also* Discretizing
  numeric attributes

instance-based learning, 128, 129

  interval, 88

linear models, 119

linear ordering, 349

mixture model, 268

1R, 86

statistical modeling, 92

numeric-attribute problem, 11

numeric prediction, 43–45, 243–254

  evaluation, 176–179

forward stagewise additive modeling, 325

linear regression, 119–120

locally weighted linear regression, 251–253
numeric prediction (continued)
  model tree, 244–251. See also model
tree
  rules, 251
  stacking, 334
trees, 76, 243

NumericToBinary, 399
NumericTransform, 397

O
O(n), 196
O(n²), 196
Obfuscate, 396, 400
object editor, 366, 381, 393
Occam’s razor, 180, 183
oil slick detection, 23
1R procedure, 84–88, 139
OneR, 408
OneRAttributeEval, 423
one-tailed, 148
online documentation, 368
Open DB, 382
optimizing performance in Weka, 417
OptionHandler, 451, 482
option nodes, 328
option trees, 328–331
orderings
  circular, 349
  partial, 349
order-independent rules, 67, 112
OrdinalClassClassifier, 418
ordinal attributes, 51
ordinal quantities, 50
orthogonal, 307
outer cross-validation, 286
outliers, 313, 342
output
  data engineering, 287–288, 315–341. See also
engineering input and output
  knowledge representation, 61–82. See also
  knowledge representation
overfitting, 86
  Bayesian clustering, 268
category utility, 261
  forward stagewise additive regression, 326
  MDL principle, 181
  multilayer perceptrons, 233
  1R, 87
  statistical tests, 30
  support vectors, 217–218
  overfitting-avoidance bias, 34
  overgeneralization, 239, 243
  overlapping hyperrectangles, 239
  overlay data, 53

P
pace regression in Weka, 410
PaceRegression, 410
paired t-test, 154, 294
pairwise classification, 123, 410
pairwise coupling, 123
pairwise plots, 60
parabola, 240
parallelization, 347
parameter tuning, 286
Part, 409
partial decision tree, 207–210
partial ordering, 51
partitioning instance space, 79
pattern recognition, 39
Percentage split, 377
perceptron
  defined, 126
  kernel, 223
  learning rule, 124, 125
  linear classification, 124–126
  multilayer, 223–226, 233
  voted, 223
perceptron learning rule, 124, 125
permutation tests, 362
PKIDiscretize, 396, 398
Poisson distribution, 268
Polygon, 389
Polyline, 389
polynomial kernel, 218
popular music, 359
postal ZIP code, 57
postpruning, 34, 192
precision, 171
predicate calculus, 82
predicting performance, 146–149. See also evaluation
predicting probabilities, 157–161
PredictionAppender, 431
prediction nodes, 329
predictive accuracy in Weka, 420
PredictiveApriori, 420
Preprocess panel, 372, 380
prepruning, 34, 192
presbyopia, 13
preventive maintenance of electromechanical devices, 25–26
principal components, 307–308
PrincipalComponents, 423
principal components analysis, 306–309
principle of multiple explanations, 183
prior knowledge, 349–351
prior probability, 90
PRISM, 110–111, 112, 213
Prism, 409
privacy, 357–358
probabilistic EM procedure, 265–266
probability-based clustering, 262–265
probability cost function, 175
probability density function, 93
programming. See Weka workbench
programming by demonstration, 360
promotional offers, 27
proportional k-interval discretization, 298
propositional calculus, 73, 82
propositional rules, 69
pruning
classification rules, 203, 205
decision tree, 192–193, 312
massive datasets, 348
model tree, 245–246
noisy exemplars, 236–237
overfitting-avoidance bias, 34
reduced-error, 203
pruning set, 202
pseudocode
basic rule learner, 111
model tree, 247–250
1R, 85
punctuation conventions, 310

Q
quadratic loss function, 158–159, 161
quadratic optimization, 217
Quinlan, J. Ross, 29, 105, 198

R
R. R. Donnelly, 28
RacedIncrementalLogitBoost, 416
race search, 295
RaceSearch, 424
radial basis function (RBF) kernel, 219, 234
radial basis function (RBF) network, 234
RandomCommittee, 415
RandomForest, 407
random forest metalearner in Weka, 416
randomization, 320–321
Randomize, 400
RandomProjection, 400
random projections, 309
RandomSearch, 424
RandomTree, 424
Ranker, 424–425
RankSearch, 424
ratio quantities, 51
RBF (Radial Basis Function) kernel, 219, 234
RBF (Radial Basis Function) network, 234
RBFNetwork, 410
real-life applications. See fielded applications
real-life datasets, 10
real-world implementations. See implementation—real-world schemes
recall, 171
recall-precision curves, 171–172
Rectangle, 389
rectangular generalizations, 80
recurrent neural networks, 233
recursion, 48
recursive feature elimination, 291, 341
reduced-error pruning, 194, 203
redundant exemplars, 236
regression, 17, 76
RegressionByDiscretization, 418
regression equation, 17
regression tree, 76, 77, 243
reinforcement learning, 38
relational data, 49
relational rules, 74
relations, 73–75
relative absolute error, 177–179
relative error figures, 177–179
relative squared error, 177, 178
RELIEF, 341
ReliefFAttributeEval, 422
religious discrimination, illegal, 35
remoteEngine.jar, 446
remote.policy, 446
Remove, 382
RemoveFolds, 400
RemovePercentage, 401
RemoveRange, 401
RemoveType, 397
RemoveUseless, 397
RemoveWithValues, 401
repeated holdout, 150
ReplaceMissingValues, 396, 398
replicated subtree problem, 66–68
REPTree, 407–408
Resample, 400, 403
residuals, 325
resubstitution error, 145
Ridor, 409
RIPPER rule learner, 205–214
ripple-down rules, 214
robo-soccer, 358
robust regression, 313–314
ROC curve, 168–171, 172
root mean-squared error, 178, 179
root relative squared error, 178, 179
root squared error measures, 177–179
rote learning, 76, 354
row separation, 336
rule
antecedent, 65
association, 69–70, 112–119
classification. See classification rules
consequent, 65
decision lists, 111–112
double-consequent, 118
exceptions, with, 70–72, 210–213
good (worthwhile), 202–205

nearest-neighbor, 78–79
numeric prediction, 251
order of (decision list), 67
partial decision trees, 207–210
propositional, 73
relational, 74
relations, and, 73–75
single-consequent, 118
trees, and, 107, 198
Weka, 408–409
rule-based programming, 82
rules involving relations, 73–75
rules with exceptions, 70–73, 210–213

S
sample problems. See example problems
sampling with replacement, 152
satellite images, evaluating, 23
ScatterPlotMatrix, 430
schemata search, 295
scheme-independent attribute selection, 290–292
scheme-specific attribute selection, 294–296
scientific applications, 28
scoring networks, 277–280, 283
SDR (Standard Deviation Reduction), 245
search bias, 33–34
search engine spam, 357
search methods in Weka, 421, 423–425
segment-challenge.arff, 389
segment-test.arff, 389
Select attributes panel, 392–393
selective Naive Bayes, 296
semantic relation, 349
semantic Web, 355
semisupervised learning, 337
sensitivity, 173
separate-and-conquer technique, 112, 200
sequential boosting-like scheme, 347
sequential minimal optimization (SMO) algorithm, 410
setOptions(), 482
sexual discrimination, illegal, 35
shapes problem, 73
sigmoid function, 227, 228
sigmoid kernel, 219
*Simple CLI*, 371, 449, 450
*SimpleKMeans*, 418–419
simple linear regression, 326
*SimpleLinearRegression*, 409
*SimpleLogistic*, 410
simplest-first ordering, 34
simplicity-first methodology, 83, 183
single-attribute evaluators in Weka, 421, 422–423
single-consequent rules, 118
single holdout procedure, 150
sister-of-relation, 46–47
*SMO*, 410
smoothing
   locally weighted linear regression, 252
   model tree, 244, 251
*SMOreg*, 410
software programs. See Weka workbench
sorting, avoiding repeated, 190
soybean data, 18–22
spam, 356–357
sparse data, 55–56
sparse instance in Weka, 401
*SparseToNonSparse*, 401
specificity, 173
specific-to-general search bias, 34
*splitData()*, 480
splitter nodes, 329
splitting
   clustering, 254–255, 257
   decision tree, 62–63
   entropy-based discretization, 301
   massive datasets, 347
   model tree, 245, 247
   subexperiments, 447
   surrogate, 247
*SpreadSubsample*, 403
squared-error loss function, 227
squared error measures, 177–179
stacked generalization, 332
stacking, 332–334
*Stacking*, 417
*StackingC*, 417
stale data, 60
standard deviation reduction (SDR), 245
standard deviations from the mean, 148
*Standardize*, 398
standardizing, 56
statistical modeling, 88–97
   document classification, 94–96
   missing values, 92–94
   normal-distribution assumption, 92
   numeric attributes, 92–94
statistics, 29–30
*Status box*, 380
step function, 227, 228
stochastic algorithms, 348
stochastic backpropagation, 232
stopping criterion, 293, 300, 326
stopwords, 310, 352
stratification, 149, 151
stratified holdout, 149
*StratifiedRemoveFolds*, 403
stratified cross-validation, 149
*StreamableFilter*, 456
string attributes, 54–55
string conversion in Weka, 399
string table, 55
*StringToNominal*, 399
*StringToWordVector*, 396, 399, 401, 462
*StripChart*, 431
structural patterns, 6
structure learning by conditional independence
tests, 280
student’s distribution with k–1 degrees of freedom, 155
student’s *t*-test, 154, 184
subexperiments, 447
subsampling in Weka, 400
subset evaluators in Weka, 421, 422
subtree raising, 193, 197
subtree replacement, 192–193, 197
success rate, 173
supervised attribute filters in Weka, 402–403
supervised discretization, 297, 298
supervised filters in Weka, 401–403
supervised instance filters in Weka, 402, 403
supervised learning, 43
support, 69, 113
support vector, 216
support vector machine, 39, 188, 214, 340
support vector machine (SVM) classifier, 341
support vector machines with Gaussian
kernels, 234
support vector regression, 219–222
surrogate splitting, 247
SVMAttributeEval, 423
SVM classifier (Support Vector Machine), 341
SwapValues, 398
SymmetricalUncertAttributeEval, 423
symmetric uncertainty, 291
systematic data errors, 59–60

T

tabular input format, 119
TAN (Tree Augmented Naïve Bayes), 279
television preferences/channels, 28–29
tenfold cross-validation, 150, 151
Tertius, 420
test set, 145
TestSetMaker, 431
text mining, 351–356
text summarization, 352
text to attribute vectors, 309–311
TextViewer, 430
TF × IDF, 311
theory, 180
threat detection systems, 357
3-point average recall, 172
threefold cross-validation, 150
ThresholdSelector, 418
time series, 311
TimeSeriesDelta, 400
TimeSeriesTranslate, 396, 399–400
timestamp, 311
TN (True Negatives), 162
tokenization, 310
tokenization in Weka, 399
top-down induction of decision trees, 105
toSource(), 453
toString(), 453, 481, 483
toy problems. See example problems
TP (True Positives), 162
training and testing, 144–146
training set, 296
TrainingSetMaker, 431
TrainTestSplitMaker, 431
transformations. See attribute transformations
transforming a multiclass problem into a two-
class one, 334–335
tree
AD (All Dimensions), 280–283
alternating decision, 329, 330, 343
ball, 133–135
decision. See decision tree
logistic model, 331
metric, 136
model, 76, 243. See also model tree
numeric prediction, 76
option, 328–331
regression, 76, 243
Tree Augmented Naïve Bayes (TAN), 279
tree classifier in Weka, 404, 406–408
tree diagrams, 82
Trees (subpackages), 451, 453
Tree Visualizer, 389, 390
ture negative (TN), 162
ture positive (TP), 162
ture positive rate, 162–163
True positive rate, 378
t-statistic, 156
t-test, 154
TV preferences/channels, 28–29
two-class mixture model, 264
two-class problem, 73
two-tailed test, 156
two-way split, 63
typographic errors, 59

U

ubiquitous data mining, 358–361
unacceptable contracts, 17
Unclassified instances, 377
Undo, 383
unit, 224
univariate decision tree, 199
universal language, 32
unlabeled data, 337–341
  clustering for classification, 337
  co-training, 339–340
  EM and co-training, 340–341
unmasking, 358
unsupervised attribute filters in Weka, 395–400
unsupervised discretization, 297–298
unsupervised instance filters in Weka, 400–401
unsupervised learning, 84
    UpdateableClassifier, 456, 482
    updateClassifier(), 482
User Classifier, 63–65, 388–391
UserClassifier, 388
user interfaces, 367–368
    Use training set, 377
utility, category, 260–262

V
validation data, 146
variance, 154, 317
Venn diagram, 81
very large datasets, 346–349
  “Very simple classification rules perform well
    on most commonly used datasets” (Holte), 88
  VFI, 414
visualization components in Weka, 430–431
    Visualize classifier errors, 387
    Visualize panel, 393
    Visualize threshold curve, 378
Vote, 417
voted perceptron, 223
VotedPerceptron, 410
voting, 315, 321, 347
voting feature intervals, 136

W
weak learners, 325
weather problem example, 10–12
  association rules for, 115–117
  attribute space for, 292–293
  as a classification problem, 42
  as a clustering problem, 43–44
  converting data to ARFF format, 370
cost matrix for, 457
evaluating attributes in, 85–86
infinite rules for, 30
item sets, 113–115
  as a numeric prediction problem, 43–44
web mining, 355–356
weight decay, 233
weighted instances, 252
WeightedInstancesHandler, 482
weighting attributes, 237–238
weighting models, 316
weka.associations, 455
weka.attributeSelection, 455
weka.classifiers, 453
weka.classifiers.bayes.NaiveBayesSimple, 472
weka.classifiers.Classifier, 453
weka.classifiers.lazy.IB1, 472
weka.classifiers.lazy.IBk, 482, 483
weka.classifiers.rules.P prism, 472
weka.classifiers.trees, 453
weka.classifiers.trees.Id3, 471, 472
weka.clusterers, 455
weka.core, 451, 452, 482–483
weka.estimators, 455
weka.filters, 455
Weka workbench, 365–483
  class hierarchy, 471–483
  classifiers, 366, 471–483
  command-line interface, 449–459. See also
    command-line interface
  elementary learning schemes, 472
  embedded machine learning, 461–469
  example application (classify text files into
    two categories), 461–469
  Experimenter, 437–447
  Explorer, 369–425. See also Explorer
  implementing classifiers, 471–483
  introduction, 365–368
  Knowledge Flow interface, 427–435
  neural-network GUI, 411
  object editor, 366
  online documentation, 368
  user interfaces, 367–368
William of Occam, 180
Winnow, 410
Winnow algorithm, 126–128
wisdom, defined, 37
Wolpert, David, 334
word conversions, 310
World Wide Web mining, 354–356
wrapper, 290, 341, 355
wrapper induction, 355
WrapperSubsetEval, 422
writing classifiers in Weka, 471–483

Z
0-1 loss function, 158
0.632 bootstrap, 152
1R method, 84–88
zero-frequency problem, 160
zero point, inherently defined, 51
ZeroR, 409
ZIP code, 57
About the Authors

Ian H. Witten is a professor of computer science at the University of Waikato in New Zealand. He is a fellow of the Association for Computing Machinery and the Royal Society of New Zealand. He received the 2004 IFIP Namur Award, a biennial honor accorded for outstanding contribution with international impact to the awareness of social implications of information and communication technology. His books include *Managing gigabytes* (1999) and *How to build a digital library* (2003), and he has written many journal articles and conference papers.

Eibe Frank is a senior lecturer in computer science at the University of Waikato. He has published extensively in the area of machine learning and sits on the editorial boards of the *Machine Learning Journal* and the *Journal of Artificial Intelligence Research*. He has also served on the programming committees of many data mining and machine learning conferences. As one of the core developers of the Weka machine learning software that accompanies this book, he enjoys maintaining and improving it.