Weka: Practical Machine Learning Tools and Techniques with Java Implementations

Ian H. Witten, Eibe Frank, Len Trigg, Mark Hall, Geoffrey Holmes, and Sally Jo Cunningham,
Department of Computer Science, University of Waikato, New Zealand.

Introduction

The Waikato Environment for Knowledge Analysis (Weka) is a comprehensive suite of Java class libraries that implement many state-of-the-art machine learning and data mining algorithms. Weka is freely available on the World-Wide Web and accompanies a new text on data mining [1] which documents and fully explains all the algorithms it contains. Applications written using the Weka class libraries can be run on any computer with a Web browsing capability; this allows users to apply machine learning techniques to their own data regardless of computer platform.

Tools are provided for pre-processing data, feeding it into a variety of learning schemes, and analyzing the resulting classifiers and their performance. An important resource for navigating through Weka is its on-line documentation, which is automatically generated from the source.

The primary learning methods in Weka are "classifiers", and they induce a rule set or decision tree that models the data. Weka also includes algorithms for learning association rules and clustering data. All implementations have a uniform command-line interface. A common evaluation module measures the relative performance of several learning algorithms over a given data set.

Tools for pre-processing the data, or "filters," are another important resource. Like the learning schemes, filters have a standardized command-line interface with a set of common command-line options.

The Weka software is written entirely in Java to facilitate the availability of data mining tools regardless of computer platform. The system is, in sum, a suite of Java packages, each documented to provide developers with state-of-the-art facilities.

Javadoc and the class library

One advantage of developing a system in Java is its automatic support for documentation. Descriptions of each of the class libraries are automatically compiled into HTML, providing an invaluable resource for programmers and application developers alike.

The Java class libraries are organized into logical packages—directories containing a collection of related classes. The set of packages is illustrated in Figure 1. They provide interfaces to pre-processing routines including feature selection, classifiers for both categorical and numeric learning tasks, meta-classifiers for enhancing the performance of classifiers (for example, boosting and bagging), evaluation according to different criteria (for example, accuracy, entropy, root-squared mean error, cost-sensitive classification, etc.) and experimental support for verifying the robustness of models (cross-validation, bias-variance decomposition, and calculation of the margin).

Weka's core

The *core* package contains classes that are accessed from almost every other class in Weka. The most important classes in it are *Attribute*, *Instance*, and *Instances*. An object of class Attribute represents an attribute—it contains the attribute's name, its type, and, in case of a nominal attribute, its possible values. An object of class Instance contains the attribute values of a particular instance; and an object of class Instances contains an ordered set of instances—in other words, a dataset.

Data Pre-Processing

Weka's pre-processing capability is encapsulated in an extensive set of routines, called *filters*, that enable data to be processed at the instance and attribute value levels. Table 1 lists the most important filter algorithms that are included.

weka.filter.AddFilter
weka.filter.DeleteFilter
weka.filter.MakeIndicatorFilter
weka.filter.MergeAttrbuteValuesFilter
weka.filter.NominalToBinaryFilter
weka.filter.SelectFilter
weka.filter.ReplaceMissingValuesFilter
weka.filter.SwapAttributeValuesFilter
weka.filter.DiscretiseFilter
weka.filter.NumericTransformFilter

Table 1: The filter algorithms in Weka

General manipulation of attributes

Many of the filter algorithms provide facilities for general manipulation of attributes. For example, the first two items in Table 1, *AddFilter* and *DeleteFilter*, insert and delete attributes. *MakeIndicatorFilter* transforms a nominal attribute into a binary indicator attribute. This is useful when

a multi-class attribute should be represented as a two-class attribute.

In some cases it is desirable to merge two values of a nominal attribute into a single value. This can be done in a straightforward way using *MergeAttributeValuesFilter*. The name of the new value is a concatenation of the two original ones.

Some learning schemes—for example, support vector machines—can only handle binary attributes. The advantage of binary attributes is that they can be treated as either being nominal or numeric. *NominalToBinaryFilter* transforms multi-valued nominal attributes into binary attributes.

SelectFilter is used to delete all instances from a dataset that exhibit one of a particular set of nominal attribute values, or a numeric value below or above a certain threshold.

One possibility of dealing with missing values is to globally replace them before the learning scheme is applied. *ReplaceMissingValuesFilter* substitutes the mean (for numeric attributes) or the mode (for nominal attributes) for each missing value.

Transforming numeric attributes

Some filters pertain specifically to numeric attributes. For example, an important filter for practical applications is the *DiscretiseFilter*. It implements an unsupervised and a supervised discretization method. The unsupervised method implements equal width binning. If the index of a class attribute is set, the method will perform supervised discretization using MDL [2].

In some applications, it is appropriate to transform a numeric attribute before a learning scheme is applied, for example, to replace each value by its square root. *NumericTransformFilter* transforms all numeric attributes among the selected attributes using a user-specified transformation function.

Feature Selection

Another essential data engineering component of any applied machine learning system is the ability to select potentially relevant features for inclusion in model induction. The Weka system provides three feature selection systems: a locally produced correlation based technique [3], the wrapper method and Relief [4].

Learning schemes

Weka contains implementations of many algorithms for classification and numeric prediction, the most important of which are listed in Table 2. Numeric prediction is interpreted as prediction of a continuous class. The *Classifier* class defines the general structure of any scheme for classification or numeric prediction.

weka.classifiers.ZeroR
weka.classifiers.OneR
weka.classifiers.NaiveBayes
weka.classifiers.DecisionTable
weka.classifiers.Ibk
weka.classifiers.j48.J48
weka.classifiers.j48.PART
weka.classifiers.SMO
weka.classifiers.LinearRegression
weka.classifiers.m5.M5Prime
weka.classifiers.LWR
weka.classifiers.DecisionStump

Table 2: The basic learning schemes in Weka

The most primitive learning scheme in Weka, *ZeroR*, predicts the majority class in the training data for problems with a categorical class value, and the average class value for numeric prediction problems. It is useful for generating a baseline

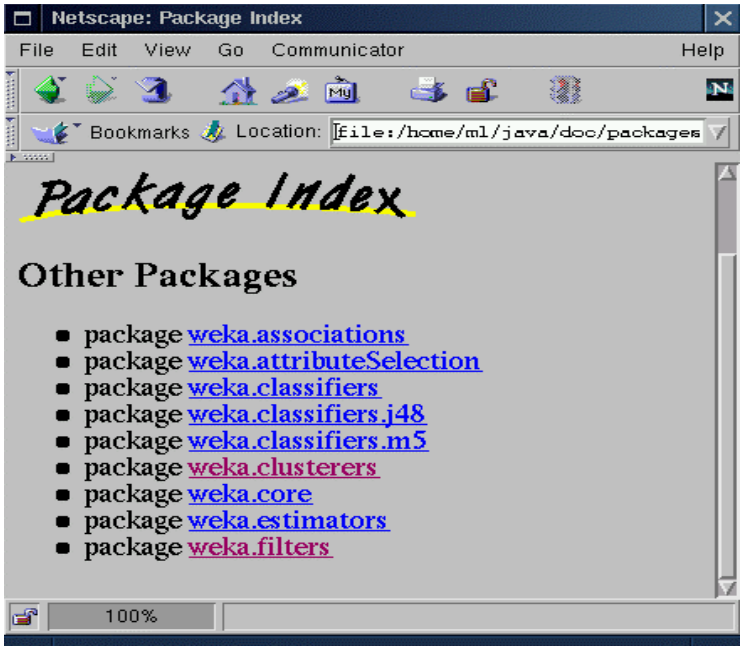

Figure 1 Package Hierarchy in Weka

performance that other learning schemes are compared to. In some cases, it is possible that other learning schemes perform worse than *ZeroR*, an indicator of substantial overfitting.

The next scheme, *OneR*, produces very simple rules based on a single attribute [5]. *NaiveBayes* implements the probabilistic Naïve Bayesian classifier. *DecisionTable* employs the wrapper method to find a good subset of attributes for inclusion in the table. This is done using a best-first search. *IBk* is an implementation of the k-nearest-neighbours classifier [6]. The number of nearest neighbours (*k*) can be set manually, or determined automatically using cross-validation.

j48 is an implementation of C4.5 release 8 [7] that produces decision trees. This is a standard algorithm that is widely used for practical machine learning. *Part* is a more recent scheme for producing sets of rules called "decision lists"; it works by forming partial decision trees and immediately converting them into the corresponding rule. *SMO* implements the "sequential minimal optimization" algorithm for support vector machines, which are an important new paradigm in machine learning [8].

The next three learning schemes in Table 2 represent methods for numeric prediction. The simplest is linear regression. *M5Prime* is a rational reconstruction of Quinlan's M5 model tree inducer [9]. *LWR* is an implementation of a more sophisticated learning scheme for numeric prediction, using locally weighted regression [10].

DecisionStump builds simple binary decision "stumps" (1-level decision trees) for both numeric and nominal classification problems. It copes with missing values by extending a third branch from the stump—in other words, by treating "missing" as a separate attribute value. *DecisionStump* is mainly used in conjunction with the *LogitBoost* boosting method, discussed in the next section.

Meta-Classifiers

Recent developments in computational learning theory have led to methods that enhance the performance or extend the capabilities of these basic learning schemes. We call these performance enhancers "meta-learning schemes" or "meta-classifiers" because they operate on the output of other learners. Table 3 summarizes the most important meta-classifiers in Weka.

The first of these schemes is an implementation of the bagging procedure [11]. This implementation allows a user to set the number of bagging iterations to be performed.

AdaBoost.M1 [12] similarly gives the user control over the boosting iterations performed. Another boosting procedure is implemented by *LogitBoost* [13], which is suited to problems involving two-class situations—for example, the *SMO* class from above. In order to apply these schemes to multi-class datasets it is necessary to transform the multi-class problem into several two-class ones, and combine the results. *MultiClassClassifier* does exactly that.

weka.classifiers.Bagging
weka.classifiers.AdaBoostM1
weka.classifiers.LogitBoost
weka.classifiers.MultiClassClassifier
weka.classifiers.CVParameterSelection

Table 3: The meta-classifier schemes in Weka

Additional learning schemes

Weka is not limited to supporting classification schemes; the class library includes representative implementations from other learning paradigms.

Association rules

Weka contains an implementation of the *Apriori* learner for generating association rules, a commonly used technique in market basket analysis [14]. This algorithm does not seek rules that predict a particular class attribute, but rather looks for any rules that capture strong associations between different attributes.

Clustering

Methods of clustering also do not seek rules that predict a particular class, but rather try to divide the data into natural groups or "clusters." Weka includes an implementation of the EM algorithm, which can be used for unsupervised learning. Like Naïve Bayes, it makes the assumption that all attributes are independent random variables.

Evaluation and Benchmarking

One of the key aspects of the Weka suite is the ability it provides to evaluate learning schemes consistently. Table 4 contains a condensed summary of the current "league table" in terms of applying the machine learning schemes to all of the datasets we have collected (37 from the UCI repository [14]). All schemes are tested by ten by ten stratified cross-validation.

W-L	Wins	Loss	Scheme
208	254	46	LogitBoost -I 100 Decision Stump
155	230	75	LogitBoost -I 10 Decision Stump
132	214	82	AdaBoostM1 Decision Trees
118	209	91	Naïve Bayes
62	183	121	Decision Trees
14	168	154	IBk Instance-based learner
-65	120	185	AdaBoostM1 Decision Stump
-140	90	230	OneR—Simple Rule learner
-166	77	243	Decision Stump
-195	9	204	ZeroR

Table 4: Ranking schemes

Column 2, *Wins*, is the number of datasets for which the scheme performed significantly better (at the 95% confidence level) than another scheme. *Loss* is the number of datasets for which a scheme performed significantly worse than another scheme. *W-L* is the difference between wins and losses to give an overall score. It would appear, for these 37 test sets, that Logit boosting simple stumps for 10 or 100 iterations is the best overall method among the schemes available in Weka.

Building Applications with Weka

In most data mining applications the machine learning component is just a small part of a far larger software system. To accommodate this, it is possible to access the programs in Weka from inside one's own code. This allows the machine learning subproblem to be solved with a minimum of additional programming.

For example, Figure 2 shows a Weka applet written to test the usability of machine learning techniques in the objective measurement of mushroom quality. Image processing a picture of a mushroom cap (at left in Figure 2) provides data for the machine learning scheme to differentiate between A, B and C grade mushrooms [15].

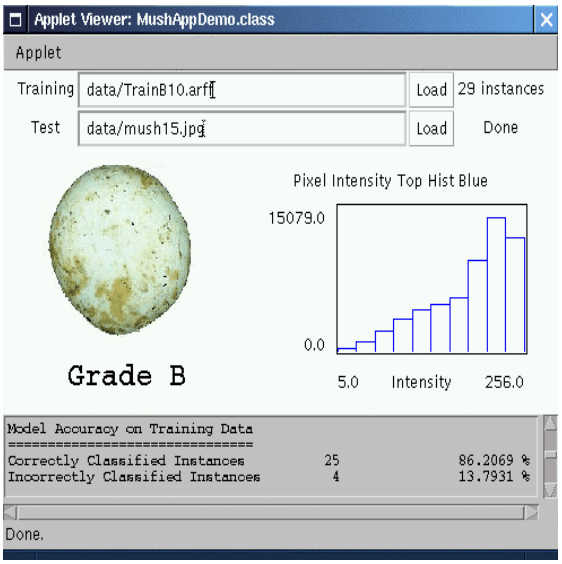

Figure 2: Mushroom grading applet

Conclusions

As the technology of machine learning continues to develop and mature, learning algorithms need to be brought to the desktops of people who work with data and understand the application domain from which it arises. It is necessary to get the algorithms out of the laboratory and into the work environment of those who can use them. Weka is a significant step in the transfer of machine learning technology into the workplace.

References

[1] Witten, I. H., and Frank E. (1999) *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, Morgan Kaufmann, San Francisco.

[2] Fayyad, U.M. and Irani, K.B. (1993) "Multi-interval discretization of continuous-valued attributes for classification learning." *Proc IJCAI*, 1022–1027. Chambery, France.

[3] Hall, M.A. and Smith, L.A. (1998) "Practical feature subset selection for machine learning." *Proc Australian Computer Science Conference*, 181–191. Perth, Australia.

[4] Kira, K. and Rendell, L.A. (1992) "A practical approach to feature selection." *Proc 9th Int Conf on Machine Learning*, 249-256.

[5] Holte, R.C. (1993) "Very simple classification rules perform well on most commonly used datasets." *Machine Learning*, Vol. 11, 63–91.

[6] Aha, D. (1992) "Tolerating noisy, irrelevant, and novel attributes in instance-based learning algorithms." *Int J Man-Machine Studies*, Vol. 36, 267–287.

[7] Quinlan, J.R. (1993) *C4.5: Programs for machine learning*. Morgan Kaufmann, San Mateo, CA.

[8] Burges, C.J.C. (1998) "A tutorial on support vector machines for pattern recognition." *Data Mining and Knowledge Discovery*, Vol. 2(1), 121-167.

[9] Wang, Y. and Witten, I.H. (1997) "Induction of model trees for predicting continuous classes." *Proc Poster Papers of the European Conference on Machine Learning*, 128-137. Prague.

[10] Atkeson, C.G., Schaal, S.A. and Moore, A.W. (1997) "Locally weighted learning." *AI Review*, Vol. 11, 11–71.

[11] Breiman, L. (1992) "Bagging predictors." *Machine Learning*, Vol. 24, 123–140.

[12] Freund, Y. and Schapire, R.E. (1996) "Experiments with a new boosting algorithm." *Proc COLT*, 209–217. ACM Press, New York.

[13] Friedman, J.H., Hastie, T. and Tibshirani, R. (1998) "Additive logistic regression: a statistical view of boosting." Technical Report, Department of Statistics, Stanford University.

[14] Agrawal, R., Imielinski, T. And Swami, A.N. (1993) "Database mining: a performance perspective." *IEEE Trans Knowledge and Data Engineering*, Vol. 5, 914–925.

[15] Kusabs N., Bollen F., Trigg L., Holmes G. and Inglis S. (1998) "Objective measurement of mushroom quality." *Proc New Zealand Institute of Agricultural Science and the New Zealand Society for Horticultural Science Annual Convention*, Hawke's Bay, New Zealand, 51.