To Appear in Tools With AI 1996

Data Mining using $\mathcal{MLC}$++
A Machine Learning Library in C++

`http://www.sgi.com/Technology/mlc`

Ron Kohavi
Data Mining and Visualization
Silicon Graphics, Inc.
2011 N. Shoreline Blvd
Mountain View, CA 94043-1389
ronnyk@engr.sgi.com

Dan Sommerfield
Data Mining and Visualization
Silicon Graphics, Inc.
2011 N. Shoreline Blvd
Mountain View, CA 94043-1389
sommda@engr.sgi.com

James Dougherty
Platform Group
Sun Microsystems MS-USUN03-306
2550 Garcia Ave.
Mountain View, CA 94043
jamesd@eng.sun.com

Abstract

Data mining algorithms including machine learning, statistical analysis, and pattern recognition techniques can greatly improve our understanding of data warehouses that are now becoming more widespread. In this paper, we focus on classification algorithms and review the need for multiple classification algorithms. We describe a system called $\mathcal{MLC}$++, which was designed to help choose the appropriate classification algorithm for a given dataset by making it easy to compare the utility of different algorithms on a specific dataset of interest. $\mathcal{MLC}$++ not only provides a workbench for such comparisons, but also provides a library of C++ classes to aid in the development of new algorithms, especially hybrid algorithms and multi-strategy algorithms. Such algorithms are generally hard to code from scratch. We discuss design issues, interfaces to other programs, and visualization of the resulting classifiers.

1 Introduction

Data warehouses containing massive amounts of data have been built in the last decade. Many organizations, however, find themselves unable to understand, interpret, and extrapolate the data to achieve a competitive advantage. Machine learning methods, statistical methods, and pattern recognition methods provide algorithms for mining such databases in order to help analyze the information, find patterns, and improve prediction accuracy.

One problem that users and analysts face when trying to uncover patterns, build predictors, or cluster data is that there are many algorithms available and it is very hard to determine which one to use. We detail a system called $\mathcal{MLC}$++, a Machine Learning library in C++ that was designed to aid both algorithm selection and development of new algorithms.

The $\mathcal{MLC}$++ project started at Stanford University in the summer of 1993 and is currently public domain software (including sources). A brief description of the library and plan was given in Kohavi, John, Long, Manley & Pfleger (1994). The distribution moved to Silicon Graphics in late 1995. Dozens of people have used it, and over 250 people are on the mailing list.

We begin the paper with the motivation for the $\mathcal{MLC}$++ library, namely the fact that there cannot be a single best learning algorithm for all tasks. This has been proven theoretically and shown experimentally. Our recommendation to users is to actually run the different algorithms. Developers can use $\mathcal{MLC}$++ to create new algorithms suitable for their specific tasks.

In the second part of the paper we describe the $\mathcal{MLC}$++

system and its dual role as a system for end-users and algorithm developers. We show a large comparison of 17 algorithms on eight large datasets for the UC Irvine repository (Murphy & Aha 1996). A study of this magnitude would be extremely hard to conduct without such a tool. With $\mathcal{MLC}$++, it was mostly a matter of CPU cycles and a few scripts to parse the output. Our study shows the behavior of different algorithms on different datasets and stresses the fact that while there are no clear winners, some algorithms are (in practice) better than others on these datasets. More importantly, for each *specific* task, it is relatively easy to choose the best algorithms to use based on accuracy estimation and other utility measures (*e.g.*, comprehensibility).

In the third part of the paper we discuss the software development process in hindsight. We were forced to make many choices on the way and briefly describe how the library evolved. We conclude with related work.

2 The Best Algorithm for the Task

In theory, there is no difference between theory and practice; In practice, there is
—Chuck Reid

Theoretical results show that there is not a single algorithm that can be uniformly more accurate than others in all domains. Although such theorems are of limited applicability in practice, very little is known about which algorithms to choose for specific problems. We claim that unless an organization has specific background knowledge that can help it choose an algorithm or tailor an algorithm based on specific needs, it should simply try a few of them and pick the best one for the task.

2.1 There is Not a Single Best Algorithm Overall

There is a theoretical result that no single learning algorithm can outperform any other when the performance measure is the expected generalization accuracy. This result, sometimes called the No Free Lunch Theorem or Conservation Law (Wolpert 1994, Schaffer 1994), is assumes that all possible targets are equally likely.

In practice, of course, the user of a data mining tool is interested in accuracy, efficiency, and comprehensibility for a specific domain, just as the car buyer is interested in power, gas mileage, and safety for specific driving conditions. Averaging an algorithm's performance over all target concepts, assuming they are all equally likely, would be like averaging a car's performance over all possible terrain types, assuming they are all equally likely. This assumption is clearly wrong in practice; for a given domain, it is clear that not all concepts are equally probable.

In medical domains, many measurements (attributes) that doctors have developed over the years tend to be independent: if the attributes are highly correlated, only one attribute will be chosen. In such domains, a certain class of learning algorithms might outperform others. For example, Naive-Bayes seems to be a good performer in medical domains (Kononenko 1993). Quinlan (1994) identifies families of *parallel* and *sequential* domains and claims that neural-networks are likely to perform well in parallel domains, while decision-tree algorithms are likely to perform well in sequential domains.

Therefore, although a single induction algorithm cannot build the most accurate classifiers in all situations, some algorithms will be clear winners in specific domains, just as some cars are clear winners for specific driving conditions. One is usually given the option to test-drive a range of cars because it is not obvious which car will be best for which purpose. The same is true for data mining algorithms. The ability to easily test-drive different algorithms was one of the factors that motivated the development of $\mathcal{MLC}$++.

2.2 Take Each Algorithm for a Test Drive

First, decide on the type of vehicle—a large luxury car or a small economy model, a practical family sedan or a sporty coupe…at this point you're ready for your first trip to a dealership—but only for a test drive…
—Consumer Reports 1996 Buying Guide: How to Buy a New Car

Organizations mine their databases for different reasons. We note a few that are relevant for classification algorithms:

Classification accuracy The accuracy of predictions made about an instance. For example, whether a customer will be able to pay a loan or whether he or she will respond to a yet another credit card offer. Using methods such as holdout, bootstrap, and cross-validation (Weiss & Kulikowski 1991, Efron & Tibshirani 1995, Kohavi 1995*b*), one can estimate the future prediction accuracy on unseen data quite well in practice.

Comprehensibility The ability for humans to understand the data and the classification rules induced by the learning algorithm. Some classifiers, such as decision rules and decision trees are inherently easier to understand than neural networks. In some domains (*e.g.*, medical), the black box approach offered by neural networks is inappropriate. In others, such as handwriting recognition, it is not as important to understand why a prediction was made so long as it is accurate.

Compactness While related to comprehensibility, one does not necessarily imply the other. A Perceptron (single neuron) might be a compact classifier, yet given an instance, it may be hard to understand the labelling process. Alternatively, a decision table (Kohavi 1995*a*) may be very large, yet labelling each instance is trivial: one simply looks it up in the table.

Training and classification time The time it takes to classify versus the training time. Some classifiers, such as neural networks are fast to classify but slow to train. Other classifiers, such as nearest-neighbor algorithms and other *lazy* algorithms (see Aha (to appear) for details), are usually fast to train but slow in classification.

Given these factors, one can define a utility function to rank different algorithms (Fayyad, Piatetsky-Shapiro & Smyth 1996). The last step is to test drive the algorithms and note their utility for *your specific domain problem.* We believe that although there are many rules of thumb for choosing algorithms, choosing a classifier should be done by testing the different algorithms, just as it is best to test-drive a car. $\mathcal{MLC}$++ is your friendly car dealer, only more honest.

3 $\mathcal{MLC}$++ for End-Users

While $\mathcal{MLC}$++ is useful for writing new algorithms, most users simply use it to test different learning algorithms. Pressure from reviewers to compare new algorithms with others led us to interface *external inducers,* which are induction algorithms written by other people. $\mathcal{MLC}$++ provides the appropriate data transformations and the same interface to these external inducers. Thus while you will not find every type of car in our dealership, we provide shuttle service to take you to many other dealerships so you can easily test their cars.

3.1 Inducers

The following induction algorithms were implemented in $\mathcal{MLC}$++:

ID3 The decision tree algorithm based on Quinlan (1986).

Nearest-neighbor The classical nearest-neighbor with options for weight setting, normalizations, and editing (Dasarathy 1990, Aha 1992, Wettschereck 1994).

Naive-Bayes A simple induction algorithm that assumes a conditional independence model of attributes given the label (Domingos & Pazzani 1996, Langley, Iba & Thompson 1992, Duda & Hart 1973, Good 1965).

OODG Oblivous read-Once Decision Graph induction algorithm described in Kohavi (1995*c*).

Lazy decision trees An algorithm for building the "best" decision tree for every test instance described in Friedman, Kohavi & Yun (1996).

1R The 1R algorithm described by Holte (1993).

Decision Table A simple lookup table. A simple algorithm that is useful with feature subset selection.

Perceptron The simple Perceptron algorithm described in Hertz, Krogh & Palmer (1991).

Winnow The multiplicative algorithm described in Littlestone (1988).

Const A constant predictor based on majority.

The following external inducers are interfaced by $\mathcal{MLC}$++:

C4.5 The C4.5 decision-tree induction by Quinlan (1993).

C4.5-rules The trees to rules induction algorithm by Quinlan (1993).

CN2 The direct rule induction algorithm by Clark & Niblett (1989) and Clark & Boswell (1991).

IB The set of Instance Based learning algorithms (Nearest-neighbors) by Aha (1992).

OC1 The Oblique decision-tree algorithm by Murthy, Kasif & Salzberg (1994).

PEBLS Parallel Exemplar-Based Learning System by Cost & Salzberg (1993).

T2 The two-level error-minimizing decision tree by Auer, Holte & Maass (1995).

Not all algorithms are appropriate for all tasks. For example, Perceptron and Winnow are limited to two-class problems, which reduces their usefulness in many problems we encounter, including those tested in Section 3.5.

3.2 Wrappers and Hybrid Algorithms

Because algorithms are encapsulated as C++ objects in $\mathcal{MLC}$++, we were able to build useful wrappers. A *wrapper* is an algorithm that treats another algorithm as a black box and acts on its output. Once an algorithm is written in $\mathcal{MLC}$++, a wrapper may be applied to it with no extra work.

The two most important wrappers in $\mathcal{MLC}$++ are accuracy estimators and feature selectors. Accuracy estimators use any of a range of methods, such as holdout, cross-validation, or bootstrap to estimate the performance of an inducer (Kohavi 1995*b*). Feature selection methods run a search using the inducer itself to determine which attributes in the database are useful for learning. The wrapper approach to feature selection automatically tailors the selection to the inducer being run (John, Kohavi & Pfleger 1994).

A voting wrapper runs an algorithm on different portions of the dataset and lets them vote on the predicted class (Wolpert 1992, Breiman 1994, Perrone 1993, Ali 1996). A discretization wrapper pre-discretizes the data, allowing algorithms that do not support continuous features (or those that do not handle them well) to work properly. A parameter optimization wrapper allows tuning the parameters of an algorithm automatically based on a search in the parameter space that optimizes the accuracy estimate with different parameters.

The following inducers are created as combinations of others:

IDTM Induction of Decision Tables with Majority. A feature subset selection wrapper on top of decision tables (Kohavi 1995*a*, Kohavi 1995*c*).

C4.5-auto Automatic parameter setting for C4.5 (Kohavi & John 1995).

FSS Naive-Bayes Feature subset selection on top of Naive-Bayes (Kohavi & Sommerfield 1995).

NBTree A decision tree hybrid with Naive-Bayes at the leaves (Kohavi 1996).

The ability to create hybrid algorithms and wrapped algorithms is a very important and powerful approach for multistrategy learning. With $\mathcal{MLC}$++ you do not have to implement two algorithms; you just have to decide on how to integrate them or wrap one around the other.

3.3 $\mathcal{MLC}$++ Utilities

The $\mathcal{MLC}$++ utilities are a set of individual executables built using the $\mathcal{MLC}$++ library. They are designed to be used by end users with little or no programming knowledge. All utilities employ a consistent interface based on options that may be set on the command line or through environment variables.

Several utilities are centered around induction. These are Inducer, AccEst, and LearnCurve. Inducer simply runs the induction algorithm of your choice on the dataset, testing the resulting classifier using a test file. AccEst estimates the performance of an induction algorithm on a dataset using any of the accuracy estimation techniques provided (holdout, cross-validation, or bootstrap). LearnCurve builds a graphical representation of the learning curve of an algorithm by running the algorithm on differently sized samples of the given dataset. The output can be displayed using Mathematica or Gnuplot.

The remaining utilities provide dataset operations. The Info utility generates descriptive statistics about a dataset, including counts of the number of attributes, class probabilities, and the number of values for each attribute. The Project utility performs the equivalent of a database's SELECT operation, allowing the user to remove attributes from a dataset. The Discretization utility converts real-valued attributes into nominal-valued attributes using any of a number of supervised discretization methods supported by the library. Finally, the Conversion utility changes multi-valued nominal attributes to local or binary encodings which may be more useful for nearest-neighbor or neural-network algorithms.

3.4 Visualization

Some induction algorithms support visual output of the classifiers. All graph-based algorithms support can display two-dimensional representations of their graphs using dot and dotty (Koutsofios & North 1994). Dotty is also capable of showing extra information at each node, such as class distributions. Figure 1 shows such a graph.

The decision tree algorithms, such as ID3, may generate output for Silicon Graphics' MineSet™ product. The Tree Visualizer provides a three-dimensional view of a decision tree with interactive fly-through capability. Figure 2 shows a snapshot of the display.

$\mathcal{MLC}$++ also provides a utility for displaying General Logic Diagrams from classifiers implemented in $\mathcal{MLC}$++. General Logic Diagrams (GLDs) are graphical projections of multi-dimensional discrete spaces onto two dimensions. They are similar to Karnaugh maps, but are generalized to non Boolean inputs and outputs. A GLD provides a way of displaying up to about ten dimensions in a graphical representation that can be understood by humans. GLDs were described in Michalski (1978) and later used in Thrun *et al.* (1991) and Wnek & Michalski (1994) to compare algorithms.

Finally, the Naive Bayes algorithm may be visualized as a three-dimensional bar chart of log probabilities. This three-dimensional view is implemented using the Virtual Reality Modeling Language (VRML 1.0) standard and can be viewed by many web browsers. The height of each bar represents the evidence in favor of a class given that a single attribute is set to a specific value. Figure 3 shows a snapshot of the visualization.

3.5 A Global Comparison

Statlog (Taylor, Michie & Spiegalhalter 1994) was a large project that compared about 20 different learning algorithms on 20 datasets. The project, funded by the ESPRIT program of the European Community, lasted from October

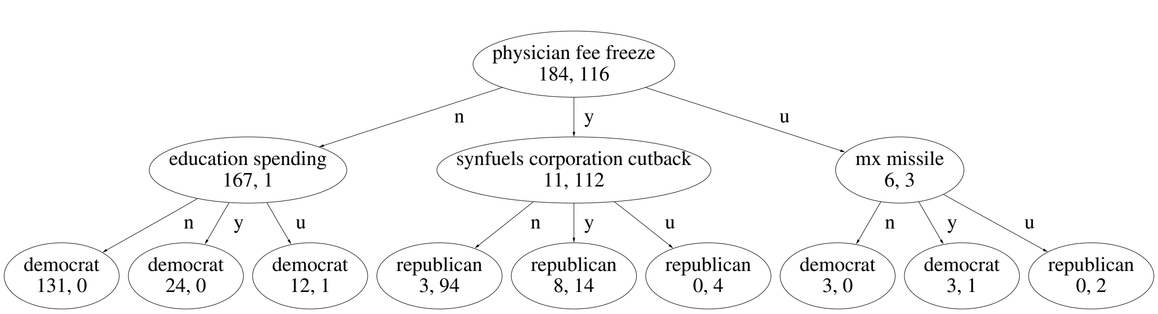


Figure 1. A dot display of a decision tree for the congressional voting dataset.

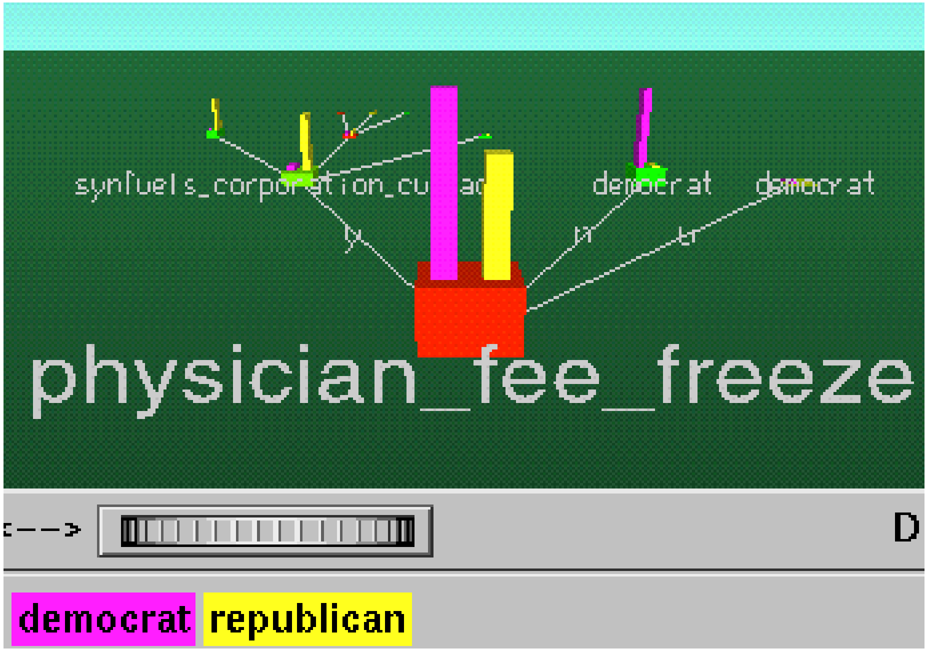


Figure 2. A snapshot of the MineSet™ Tree Visualizer fly-through for a decision tree.

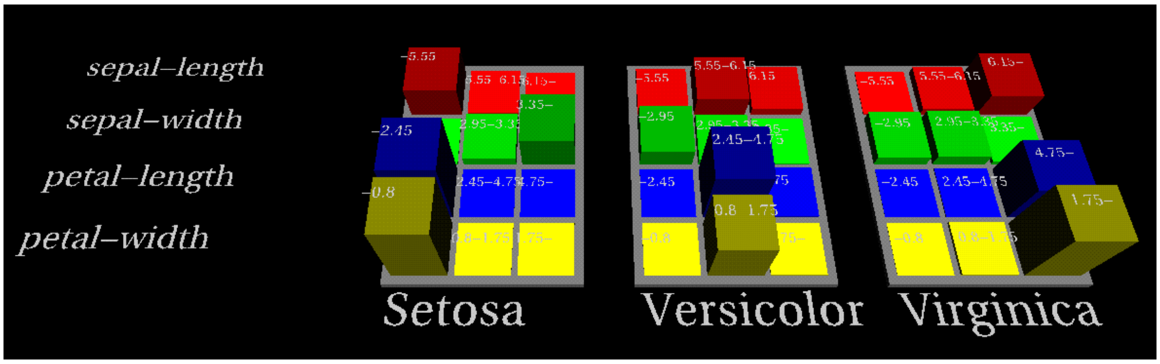


Figure 3. A snapshot of the Naive-Bayes view.

1990 to June 1993. Without a tool such as $\mathcal{MLC}$++, an organization with specific domain problems cannot easily repeat such an effort in order to choose the appropriate algorithms. Converting data formats, learning the different algorithms, and running many of them is usually out of the question. $\mathcal{MLC}$++, however, can easily provide such a comparison.

In this section we present a comparison on some large datasets from the UC Irvine repository (Murphy & Aha 1996). We also take this opportunity to correct some misunderstandings of results in Holte (1993). Table 1 shows the basic characteristics of the chosen domains. Instances with unknown values were removed from the original datasets.

Holte showed that for many small datasets commonly used by researchers in machine learning circa 1990, simple classifiers perform surprisingly well. Specifically, Holte proposed an algorithm called 1R, which learns a complex rule (with many intervals), but using only a single attribute. He claimed that such a simple rule performed surprisingly well. On sixteen datasets from the UCI repository, the error rate of 1R was 19.82% while that of C4.5 was 14.07%, a difference of 5.7%.

While it is surprising how much one can do with a single attribute, a different way to look at this result is to look at the *relative error*, or the increase in error of 1R over C4.5. This increase is over 40%, which is very significant if an organization has to pay a large amount of money for every mistake made.

Figures 4 and 5 show a comparison of 17 algorithms on eight large datasets from the UC Irvine repository. Our results show that for different domains different algorithms perform differently and that there is no single clear winner. However, they also show that for the larger real-world datasets at UC Irvine, some algorithms are generally safe-bets and some are pretty bad in general. Specifically, algorithms 2 (C4.5-auto), 8 (NBTree), and 11 (FSS Naive-Bayes) were among the best performers in three out of the eight datasets, while algorithms 6 (T2) and 7 (1R) were consistently worst performers. In fact, T2 was in the worst set in five out of eight datasets and could not even be run on two other datasets because it required over 512MB of memory.

While there is very little theory on how to select algorithms in advance, simply running many algorithms and looking at the output and accuracy is a practical solution with $\mathcal{MLC}$++.

4 $\mathcal{MLC}$++ for Software Developers

$\mathcal{MLC}$++ source code is public domain and can be used freely in both research and commercial settings. $\mathcal{MLC}$++ contains over 60,000 lines of code, 14,000 lines of regression testing code. The $\mathcal{MLC}$++ utilities are only 2,000 lines of code that use the library itself.

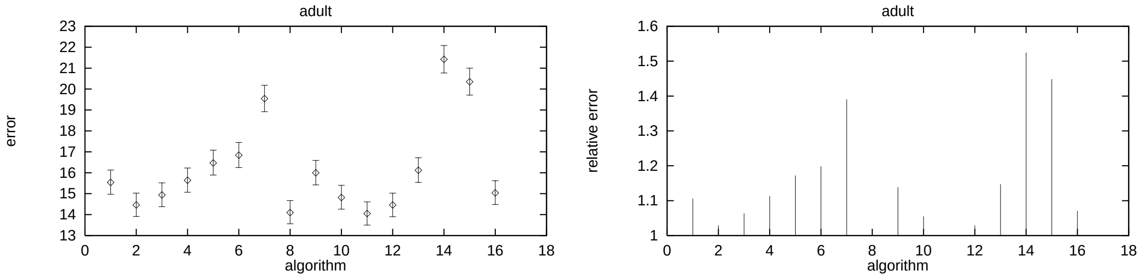
One of the advantages of using $\mathcal{MLC}$++ for software developers is that it provides high-quality code and encourages high-quality code development. By providing a large number of general purpose classes as well as a large set of classes specific to machine learning, $\mathcal{MLC}$++ ensures that coding remains a small part of the development process. Mature coding standards and a shallow class hierarchy insure that new code is robust and helpful to future developers. Although it can take longer to write fully-tested code using $\mathcal{MLC}$++, ultimately, $\mathcal{MLC}$++ greatly decreases the time needed to complete a robust product.

$\mathcal{MLC}$++ includes a library of general purpose classes, independent more machine learning, called MCore. MCore classes include strings, lists, arrays, hash tables, and input/output streams, as well as some built-in mechanisms for dealing with C++ initialization order, options, temporary file generation and cleanup, as well as interrupt handling. MCore classes not only provide a wide range of functions, they provide solid tests of these functions, reducing the time it takes to build more complex operations. All general-purpose classes used within the full library are from MCore, with the exception of graph classes from the LEDA library (Naeher 1996). Although the classes in MCore were built for use by the $\mathcal{MLC}$++ library, they are not tied to machine learning and may be used as a general purpose library. MCore is a separate library which may be linked independently of the whole library.

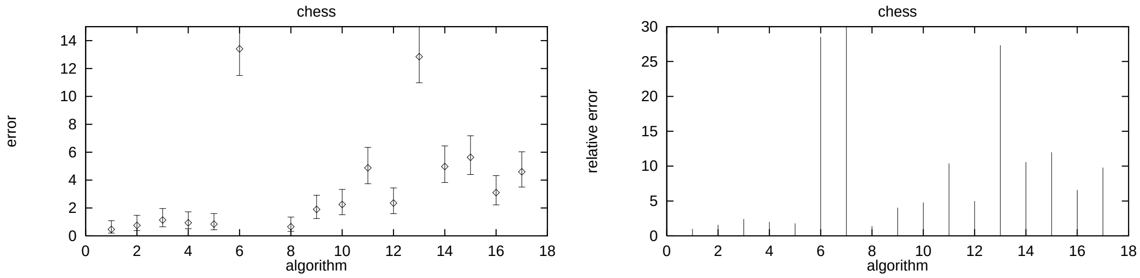
$\mathcal{MLC}$++ classes are arranged more or less as a library of independent units, rather than as a tree or forest of mutually-referent set of modules requiring multiple link and compile stages. The ML and MInd modules follow this philosophy by providing a series of classes which encapsulate basic concepts in machine learning. Inheritance is only used where it helps the programmer. By resisting the temptation to maximize use of C++ objects, we were able to provide a set of basic classes with clear-cut interfaces for use by anyone developing a machine learning algorithm.

The important concepts provided by the library include Instance lists, Categorizers (classifiers), Inducers, and Discretizers. Instance lists and supporting classes hold the database on which learning algorithms are run. Categorizers (also known as classifiers) and Inducers (induction algorithms) represent the algorithms themselves: an Inducer is a machine learning algorithm that produces a classifier that, in turn, assigns a class to each instance. Discretizers replace real-valued attributes in the database with discrete-valued attributes for algorithms which can only use discrete values. $\mathcal{MLC}$++ provides several discretization methods (Dougherty, Kohavi & Sahami 1995).

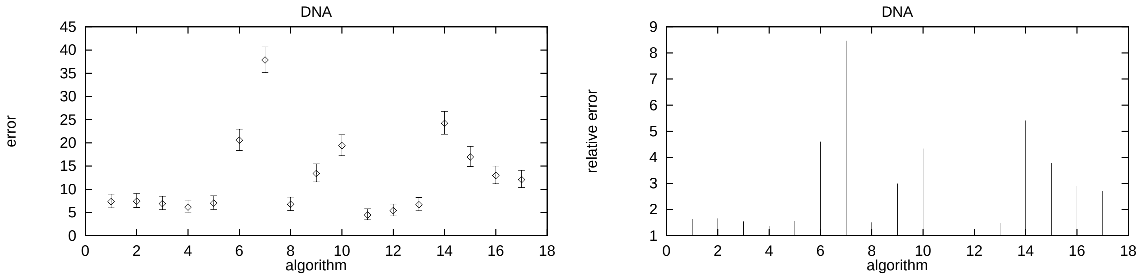
Programming with $\mathcal{MLC}$++ generally requires little coding. Because $\mathcal{MLC}$++ contains many well-tested modules and utility classes, the bulk of $\mathcal{MLC}$++ programming is determining how to use the existing code base to implement new



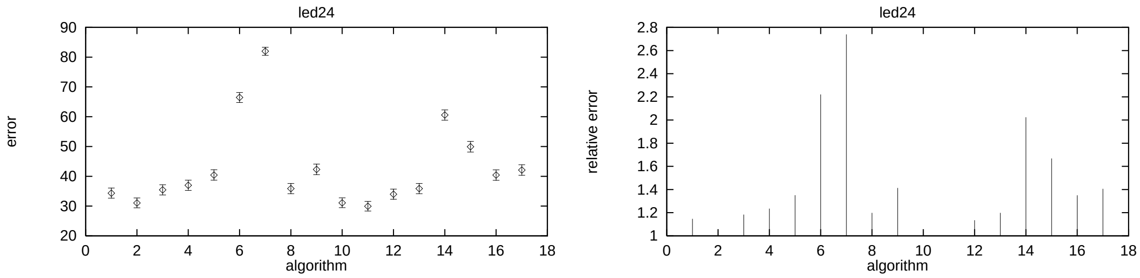
Best algorithms: 2,8,11,12, worst algorithms: 7,14,15



Best algorithms: 1,2,8, worst algorithms: 6,7,13



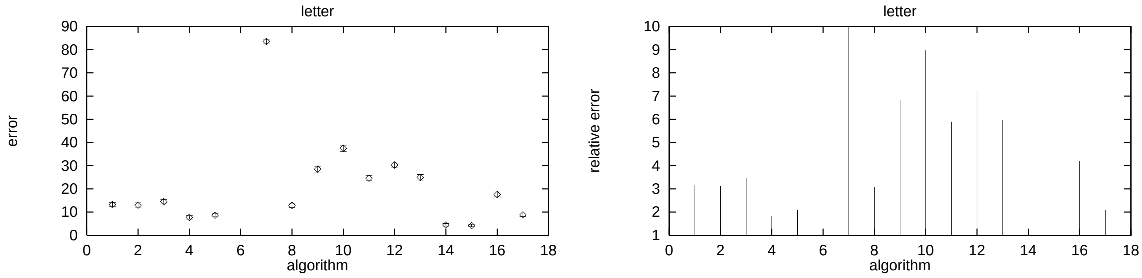
Best algorithms: 11,12, worst algorithms: 6,7,14



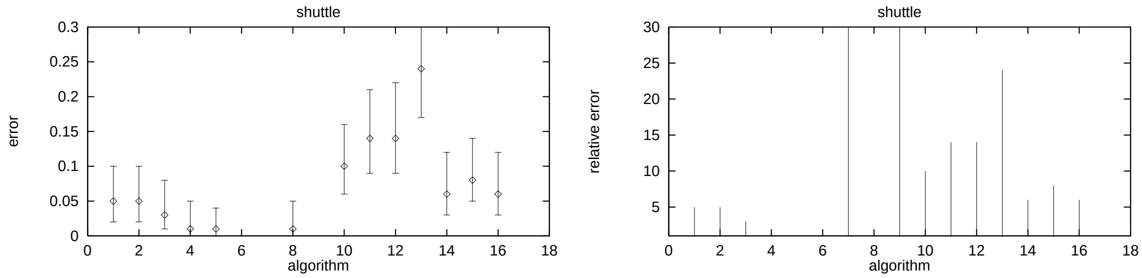
Best algorithms: 2,10,11, worst algorithms: 6,7,14

1	C4.5	2	C4.5-auto	3	C4.5 rules
4	Voted ID3 (0.6)	5	Voted ID3 (0.8)	6	T2
7	1R	8	NBTree	9	CN2
10	HOODG	11	FSS Naive Bayes	12	IDTM (Decision table)
13	Naive-Bayes	14	Nearest-neighbor (1)	15	Nearest-neighbor (3)
16	OC1	17	PEBLS		

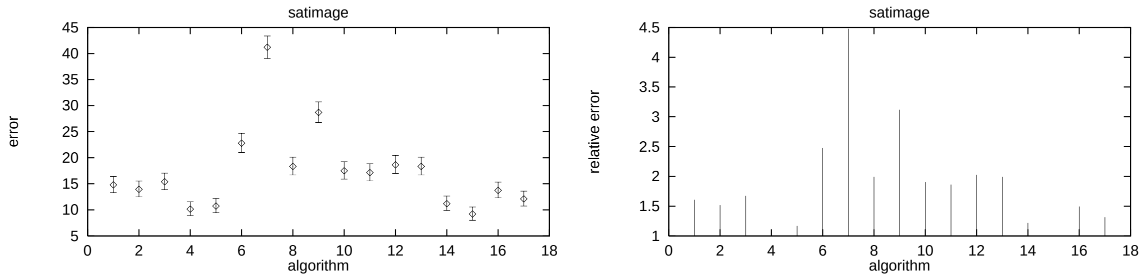
Figure 4. Error and relative errors for the learning algorithms. PEBLS (17), an interfaced algorithm, crashed on the adult dataset.



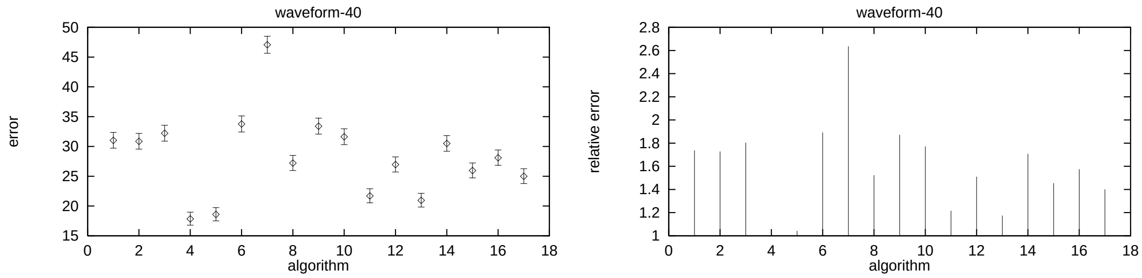
Best algorithms: 14,15, worst algorithms: 7,10,12



Best algorithms: 4,5,8, worst algorithms: 7,9,13 (7,9 not shown on left, they are out of range)



Best algorithms: 4,5,15, worst algorithms: 6,7,9



Best algorithms: 4,5, worst algorithms: 6,7,9

1	C4.5	2	C4.5-auto	3	C4.5 rules
4	Voted ID3 (0.6)	5	Voted ID3 (0.8)	6	T2
7	1R	8	NBTree	9	CN2
10	HOODG	11	FSS Naive Bayes	12	IDTM (Decision table)
13	Naive-Bayes	14	Nearest-neighbor (1)	15	Nearest-neighbor (3)
16	OC1	17	PEBLS		

Figure 5. Error and relative errors for the learning algorithms. PEBLS (17) crashed on the shuttle dataset; T2 (6) needed over 512MB of memory for the letter and shuttle datasets.

Table 1. The datasets used, the number of attributes, the training and test-set sizes, and the baseline accuracy (majority class).

Dataset	Attributes	Train size	Test size	Majority accuracy	Dataset	Attributes	Train size	Test size	Majority accuracy
adult	14	30,162	15,060	75.22	chess	36	2,130	1,066	52.22
DNA	180	2,000	1,186	51.91	led24	24	200	3000	10.53
letter	16	15,000	5,000	4.06	shuttle	9	43,500	14,500	78.60
satimage	36	4,435	2,000	23.82	waveform-40	40	300	4,700	33.84

functionality. Although learning how to use a large code base takes time, we now find that the code we write is much easier to test and debug because we can leverage the testing and debugging done on the rest of the library. Additionally, forcing the developer to break a problem down into $\mathcal{MLC}$++ concepts insures that $\mathcal{MLC}$++ maintains its consistent design, which in turn insures consistent code quality.

Each class in $\mathcal{MLC}$++ is testable almost independently with little reliance on other classes. We have built a large suite of regression tests to insure the quality of each class in the library. The tests will immediately determine if a code change has created a problem. The goal of these tests is to force bugs to show up as close to their sources as possible. Likewise, porting the library is greatly simplified by these tests, which provide a clear indication of any problems with a port. Unfortunately, many C++ compilers still do not support full ANSI C++ templates that are required to compile $\mathcal{MLC}$++.

5 $\mathcal{MLC}$++ Development in Hindsight

Hindsight is always twenty-twenty
—Billy Wilder

Many decisions had to be made during the development of $\mathcal{MLC}$++. We tackled such issues as which language to use, what libraries, whether users must know how to code in order to use it, and how much to stress efficiency versus safety.

We decided that $\mathcal{MLC}$++ should be useful to end-users who have neither programmed before nor want to program now (many lisp packages for machine learning require writing code just to load the data). We realized that a GUI would be very useful; however, we concentrated on adding functionality, providing only a simple but general environment variable-based command-line interface. We also provided the source code for other people to interface. $\mathcal{MLC}$++ was used to teach machine learning courses at Stanford and many changes to the library were made based on student's feedback.

We chose C++ for a number of reasons. First, C++ is a good general-purpose language; it has many useful features but imposes no style requirements of its own. Features such as templates allowed us to maintain generality and reusability along with the safety of strong typing and static checking. Second, the language's object-oriented features allowed us to decompose the library into a set of modular objects which could be tested independently. At the same time, C++'s relative flexibility allowed us to avoid the object-oriented paradigm when needed. Third, C++ has no language-imposed barriers to efficiency; functions can be made inline, and objects and be placed on the stack for faster access. This gave use the ability to optimize code in critical sections. Finally, C++ is a widely accepted language which is gaining popularity quickly. Since the project was created at Stanford, it was extremely useful to use a language which everybody wanted to learn and use.

We also made a decision to use the best compiler we found at the time (CenterLine's ObjectCenter) and to use all available features. We assumed that GNU's g++ compiler would catch up. In 1987 "Mike Tiemann gave a most animated and interesting presentation of how the GNU C++ compiler he was building would do just about everything and put all other C++ compiler writers out of business" (Stroustroup 1994). Sadly, the GNU C++ compiler is still weaker than most commercial grade compilers and (at least as of 1995) cannot handle templates well enough. Moreover, most PC-based compilers still cannot compile $\mathcal{MLC}$++. We hope that this will change as 32-bit operating systems mature and compilers improve.

We quickly chose LEDA (Naeher 1996) for graph manipulation and algorithms, and GNU's libg++, which provided some basic data structures. Unfortunately, the GNU library was found to be deficient in many respects. It hasn't kept up with the emerging C++ standards (*e.g.*, constness issues, templates) and we slowly rewrote all the classes used. Today $\mathcal{MLC}$++ does not use libg++. If we were starting the project today, we would likely use the C++ Standard Template Library (STL), since it provides a large set of solid data structures and algorithms, has the static safety of templates, and is likely to be a widely accepted standard. One current disadvantage of many class libraries, including $\mathcal{MLC}$++, is that developers must learn a large code base of standard data structures and algorithms. This greatly increases the learning curve to use the library. Use of a standard library like STL might flatten that curve.

Much of what was done in $\mathcal{MLC}$++ was motivated by research interests of the first author. This resulted in a skewed library with many symbolic algorithms and no work on neural networks nor any statistical regression algorithms. Neural network and statistical algorithms were also shunned because many such algorithms already existed and there seemed to be little need to write yet another one when it would not contribute to immediate research interests. However, today, with the focus shifting toward data mining, the library is becoming increasingly more balanced.

6 Related Work

Several large efforts have been made to describe data mining algorithms for knowledge discovery in data. We refer the reader to the URL
`http://info.gte.com/~kdd/siftware.html`
for pointers and description. We briefly mention systems that provide multiple tools or that have goals similar to $\mathcal{MLC}$++.

MineSet™ is Silicon Graphics' data mining and visualization product. Release 1.1 uses $\mathcal{MLC}$++ as a base for the induction and classification algorithms. Classification models built are then shown using the 3D visualization tools. MineSet™ has a GUI interface and accesses commercial databases including Oracle, Sybase, and Informix. Besides classification, it also provides an association algorithm.

MLToolbox is a collection of many publicly available algorithms and re-implementations of others. The algorithms do not share common code and interface.

TOOLDIAG is a collection of methods for statistical pattern recognition, especially classification. While it contains many algorithm, such as k-nearest neighbor, radial basis functions, parzen windows, feature selection and extraction, it is limited to continuous attributes with no missing values.

Mobal is a multistrategy tool which integrates manual knowledge acquisition techniques with several automatic first-order learning algorithms. The entire system is rule-based and existing knowledge is stored using a syntax similar to predicate logic.

Emerald is a research prototype from George Mason University. Its main features are five learning programs and a GUI. The lisp-based system is capable of learning rules which are then translated to English and spoken by a speech synthesizer. The algorithms include algorithms for learning rules from examples, learning structural descriptions of objects, conceptually grouping objects or events, discovering rules characterizing sequences, and learning equations based on qualitative and quantitative data.

Sipina-W is a machine learning and knowledge engineering tool package which implemented CART, ID3, C4.5, Elisee (segmentation), Chi^{2}Aid, and SIPINA algorithms for classification. Sipina-W runs on real or discrete-valued data sets and is oriented toward the building and testing of expert systems. It reads ASCII, dBase, and Lotus format files.

Commercial tools for data mining include IBM's Intelligent Data Miner, Clementine, and Darwin. Except for Darwin, they do not share a common underlying library. IBM's Intelligent Data Miner provides a variety of knowledge discovery algorithms for classification, associations, clustering, and sequential pattern discovery. Clementine includes neural network and an interface to C4.5 for decision tree and rule induction. It has a strong, dataflow-based GUI and simple visualizations. Darwin is a suite of learning tools developed by Thinking Machines. It is intended for use on large databases and uses parallel processing for speedup. Its algorithms include Classification and Regression Trees (CART), Neural networks, Nearest Neighbor, and Genetic Algorithms. Darwin also includes some 2D visualization.

7 Summary

$\mathcal{MLC}$++, a Machine Learning library in C++, has greatly evolved over the last three years. It now provides developers with a substantial piece of code that is well-organized into a useful C++ hierarchy. Even though (or maybe because) it was mostly a research project, we managed to keep the code quality high with many regression tests.

The library provides end-users with the ability to easily test-drive different induction algorithms on datasets of interest. Accuracy estimation and visualization of classifiers allow one to pick the best algorithm for the task.

Silicon Graphics new data mining product, MineSet™ 1.1, has classifiers built on top of $\mathcal{MLC}$++ with a GUI and interfaces to commercial databases. We hope this will open machine learning and data mining to a wider audience.

8 Acknowledgments

The $\mathcal{MLC}$++ project started in the summer of 1993 and continued for two years at Stanford University. Since late 1995 the distribution and support have moved to Silicon Graphics. $\mathcal{MLC}$++ is public domain and can be used by anyone for any purpose. Nils Nilsson and Yoav Shoham provided support for this project at its initial stages. Wray Buntine, George John, Pat Langley, Ofer Matan, Karl Pfleger, and Scott Roy contributed to the design of $\mathcal{MLC}$++. Many students at Stanford have worked on $\mathcal{MLC}$++, including: Robert Allen, Brian Frasca, James Dougherty, Steven Ihde, Ronny Kohavi, Alex Kozlov, Jamie Chia-Hsin Li, Richard Long, David Manley, Svetlozar Nestorov, Mehran Sahami, Dan Sommerfield, and Yeogirl Yun. $\mathcal{MLC}$++ was partly funded by ONR grants N00014-94-1-0448, N00014-95-1-0669, and NSF grant IRI-9116399. Eric Bauer, Clayton Kunz are now working on extending $\mathcal{MLC}$++ at Stanford under support from Silicon Graphics. Ronny Kohavi and Dan Sommerfield continue to work on this project at Silicon Graphics.

References

Aha, D. W. (1992), 'Tolerating noisy, irrelevant and novel attributes in instance-based learning algorithms', *International Journal of Man-Machine Studies* **36**(1), 267–287.

Aha, D. W. (to appear), AI review journal: Special issue on lazy learning.

Ali, K. M. (1996), Learning Probabilistic Relational Concept Descriptions, PhD thesis, University of California, Irvine. http://www.ics.uci.edu/~ali.

Auer, P., Holte, R. & Maass, W. (1995), Theory and applications of agnostic PAC-learning with small decision trees, *in* A. Prieditis & S. Russell, eds, 'Machine Learning: Proceedings of the Twelfth International Conference', Morgan Kaufmann Publishers, Inc.

Breiman, L. (1994), Bagging predictors, Technical Report Statistics Department, University of California at Berkeley.

Clark, P. & Boswell, R. (1991), Rule induction with CN2: Some recent improvements, *in* Y. Kodratoff, ed., 'Proceedings of the fifth European conference (EWSL-91)', Springer Verlag, pp. 151–163.

Clark, P. & Niblett, T. (1989), 'The CN2 induction algorithm', *Machine Learning* **3**(4), 261–283.

Cost, S. & Salzberg, S. (1993), 'A weighted nearest neighbor algorithm for learning with symbolic features', *Machine Learning* **10**(1), 57–78.

Dasarathy, B. V. (1990), *Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques*, IEEE Computer Society Press, Los Alamitos, California.

Domingos, P. & Pazzani, M. (1996), Beyond independence: conditions for the optimality of the simple bayesian classifier, *in* L. Saitta, ed., 'Machine Learning: Proceedings of the Thirteenth International Conference', Morgan Kaufmann Publishers, Inc., pp. 105–112.

Dougherty, J., Kohavi, R. & Sahami, M. (1995), Supervised and unsupervised discretization of continuous features, *in* A. Prieditis & S. Russell, eds, 'Machine Learning: Proceedings of the Twelfth International Conference', Morgan Kaufmann, pp. 194–202.

Duda, R. & Hart, P. (1973), *Pattern Classification and Scene Analysis*, Wiley.

Efron, B. & Tibshirani, R. (1995), Cross-validation and the bootstrap: Estimating the error rate of a prediction rule, Technical Report 477, Stanford University, Statistics department.

Fayyad, U. M., Piatetsky-Shapiro, G. & Smyth, P. (1996), From data mining to knowledge discovery: An overview, *in* 'Advances in Knowledge Discovery and Data Mining', AAAI Press and the MIT Press, chapter 1, pp. 1–34.

Friedman, J., Kohavi, R. & Yun, Y. (1996), Lazy decision trees, *in* 'Proceedings of the Thirteenth National Conference on Artificial Intelligence', AAAI Press and the MIT Press.

Good, I. J. (1965), *The Estimation of Probabilities: An Essay on Modern Bayesian Methods*, M.I.T. Press.

Hertz, J., Krogh, A. & Palmer, R. G. (1991), *Introduction to the Theory of Neural Computation*, Addison Wesley.

Holte, R. C. (1993), 'Very simple classification rules perform well on most commonly used datasets', *Machine Learning* **11**, 63–90.

John, G., Kohavi, R. & Pfleger, K. (1994), Irrelevant features and the subset selection problem, *in* 'Machine Learning: Proceedings of the Eleventh International Conference', Morgan Kaufmann, pp. 121–129.

Kohavi, R. (1995*a*), The power of decision tables, *in* N. Lavrac & S. Wrobel, eds, 'Proceedings of the European Conference on Machine Learning', Lecture Notes in Artificial Intelligence 914, Springer Verlag, Berlin, Heidelberg, New York, pp. 174–189.

Kohavi, R. (1995*b*), A study of cross-validation and bootstrap for accuracy estimation and model selection, *in* C. S. Mellish, ed., 'Proceedings of the 14th International Joint Conference on Artificial Intelligence', Morgan Kaufmann Publishers, Inc., pp. 1137–1143.

Kohavi, R. (1995*c*), Wrappers for Performance Enhancement and Oblivious Decision Graphs, PhD thesis, Stanford University, Computer Science department. STAN-CS-TR-95-1560,
ftp://starry.stanford.edu/pub/ronnyk/teza.ps.

Kohavi, R. (1996), Scaling up the accuracy of naive-bayes classifiers: a decision-tree hybrid, *in* 'Proceedings of the Second International Conference on Knowledge Discovery and Data Mining', p. to appear.

Kohavi, R. & John, G. (1995), Automatic parameter selection by minimizing estimated error, *in* A. Prieditis & S. Russell, eds, 'Machine Learning: Proceedings of the Twelfth International Conference', Morgan Kaufmann Publishers, Inc., pp. 304–312.

Kohavi, R., John, G., Long, R., Manley, D. & Pfleger, K. (1994), MLC++: A machine learning library in C++, *in* 'Tools with Artificial Intelligence', IEEE Computer Society Press, pp. 740–743. `http://www.sgi.com/Technology/mlc`.

Kohavi, R. & Sommerfield, D. (1995), Feature subset selection using the wrapper model: Overfitting and dynamic search space topology, *in* 'The First International Conference on Knowledge Discovery and Data Mining', pp. 192–197.

Kononenko, I. (1993), 'Inductive and bayesian learning in medical diagnosis', *Applied Artificial Intelligence* **7**, 317–337.

Koutsofios, E. & North, S. C. (1994), Drawing graphs with dot. Available by anonymous ftp from `research.att.com:dist/drawdag/dotdoc.ps.Z`.

Langley, P., Iba, W. & Thompson, K. (1992), An analysis of bayesian classifiers, *in* 'Proceedings of the tenth national conference on artificial intelligence', AAAI Press and MIT Press, pp. 223–228.

Littlestone, N. (1988), 'Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm', *Machine Learning* **2**, 285–318.

Michalski, R. S. (1978), A planar geometric model for representing multidimensional discrete spaces and multiple-valued logic functions, Technical Report UIUCDCS-R-78-897, University of Illinois at Urbaba-Champaign.

Murphy, P. M. & Aha, D. W. (1996), UCI repository of machine learning databases. `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

Murthy, S. K., Kasif, S. & Salzberg, S. (1994), 'A system for the induction of oblique decision trees', *Journal of Artificial Intelligence Research* **2**, 1–33.

Naeher, S. (1996), *LEDA: A Library of Efficient Data Types and Algorithms*, 3.3 edn, Max-Planck-Institut fuer Informatik, IM Stadtwald, D-66123 Saarbruecken, FRG. `http://www.mpi-sb.mpg.de/LEDA/leda.html`.

Perrone, M. (1993), Improving regression estimation: averaging methods for variance reduction with extensions to general convex measure optimization, PhD thesis, Brown University, Physics Dept.

Quinlan, J. R. (1986), 'Induction of decision trees', *Machine Learning* **1**, 81–106. Reprinted in Shavlik and Dietterich (eds.) *Readings in Machine Learning*.

Quinlan, J. R. (1993), *C4.5: Programs for Machine Learning*, Morgan Kaufmann Publishers, Inc., Los Altos, California.

Quinlan, J. R. (1994), Comparing connectionist and symbolic learning methods, *in* S. J. Hanson, G. A. Drastal & R. L. Rivest, eds, 'Computational Learning Theory and Natural Learning Systems', Vol. I: Constraints and Prospects, MIT Press, chapter 15, pp. 445—456.

Schaffer, C. (1994), A conservation law for generalization performance, *in* 'Machine Learning: Proceedings of the Eleventh International Conference', Morgan Kaufmann Publishers, Inc., pp. 259–265.

Stroustroup, B. (1994), *The Design and Evolution of C++*, Addison-Wesley Publishing Company.

Taylor, C., Michie, D. & Spiegalhalter, D. (1994), *Machine Learning, Neural and Statistical Classification*, Paramount Publishing International.

Thrun *et al.* (1991), The Monk's problems: A performance comparison of different learning algorithms, Technical Report CMU-CS-91-197, Carnegie Mellon University.

Weiss, S. M. & Kulikowski, C. A. (1991), *Computer Systems that Learn*, Morgan Kaufmann Publishers, Inc., San Mateo, CA.

Wettschereck, D. (1994), A Study of Distance-Based Machine Learning Algorithms, PhD thesis, Oregon State University.

Wnek, J. & Michalski, R. S. (1994), 'Hypothesis-driven constructive induction in AQ17-HCI : A method and experiments', *Machine Learning* **14**(2), 139–168.

Wolpert, D. H. (1992), 'Stacked generalization', *Neural Networks* **5**, 241–259.

Wolpert, D. H. (1994), The relationship between PAC, the statistical physics framework, the Bayesian framework, and the VC framework, *in* D. H. Wolpert, ed., 'The Mathemtatics of Generalization', Addison Wesley.