Decision Tree Toolkit: A Component-Based Library of Decision Tree Algorithms

Nikos Drossos, Athanasios Papagelis, and Dimitris Kalles

Computer Technology Institute, Patras, Greece
{drosnick,papagel,kalles}@cti.gr

Abstract. This paper reports on the development of a library of decision tree algorithms in Java. The basic model of a decision tree algorithm is presented and then used to justify the design choices and system architecture issues. The library has been designed for flexibility and adaptability. Its basic goal was an open system that could easily embody parts of different conventional as well as new algorithms, without the need of knowing the inner organization of the system in detail. The system has an integrated interface (ClassExplorer), which is used for controlling and combining components that comprise decision trees. The ClassExplorer can create objects "on the fly", from classes unknown during compilation time. Conclusions and considerations about extensions towards a more visual system are also described.

1. Introduction

Decision Trees -one of the major Machine Learning (ML) paradigms- have numerous advantages over other concept learners. First, they require relatively small computational power to create a model of the underlying hypothesis and this model requires a small amount of memory to be represented; this means that they can be efficiently used over relatively big amounts of information. Second, they offer a comprehensible way to organise acquired knowledge, in contrast with other learners (like Neural Networks or NaïveBayes); thus, they provide insight on the problem rather than simple classifications/predictions. Third, the classification accuracy of the underlying hypothesis obtained by decision trees is competitive with that of most concept learners. All these justify why decision trees are so popular among researchers and many ML-oriented business applications.

Due to their success and their heuristic nature, many researchers have concentrated on the problem of improving decision tree learners. Those efforts have resulted in dozens of different methods for (mainly): pre-processing data, selecting splitting attributes, pre- and post-pruning the tree, as well many different methodologies for estimating the superiority of one algorithm over the other (*k*-fold cross-validation, leave-one-out). However, no algorithm clearly outperforms all others in all cases.

Traditionally, every researcher wanting to try out new ideas would have to create a new learner from scratch, even though most of the tasks (like reading data and creating an internal representation) were indifferent to the researcher and irrelevant with the very heart of the new algorithm. Even when portions of previous programs were used there was still the great burden of code adaptation.

D.A. Zighed, J. Komorowski, and J. Zytkow (Eds.): PKDD 2000, LNAI 1910, pp. 381−387, 2000.

The above, combined with the imperative need for smaller production times, suggest the use of reusable components that can be efficiently altered to match individual needs. This paper presents such a fully object-oriented components library that has been built with the researcher in mind [1]. This library can substantially reduce the time of deriving a new decision tree algorithm by providing building blocks of the "no-need-to-be-changed" portions of the algorithm while at the same time offering an established base where algorithms can be tested and compared to each other. Furthermore, this library is accompanied with a general tool (the ClassExplorer) where someone can easily create objects (in an interactive way) and combine them to produce a desired behaviour without having to write-compile an entire class or program.

Libraries of components are not a new concept and that holds true for Machine Learning too. The two most known ML libraries are MLC++ [2] and WEKA [3]. Both contain common induction algorithms, such as ID3 [4], C4.5 [5], *k*-Nearest Neighbors [6], NaïveBayes [7] and Decision Tables [8] written under a single framework. Moreover, they contain wrappers to wrap around algorithms. These include: feature selection, discretisation filters, bagging/combining classifiers, and more. Finally, they contain common accuracy estimation methods.

This work will not replace those established and global machine-learning tools. This library has a more limited scope: it focuses on a specific problem, the creation of binary decision trees. Reducing the scope provides a more solid framework for the specific problem. Furthermore, it reduces the complexity of the environment and the needed time for someone to familiarise himself with it. Moreover, the organization of the library is **component** and **not algorithm** oriented. This means, an easy way to interchange "building blocks" between different decision trees implementation, and thus, better chance to reveal the underlying causes of diversity in performance. Finally, the added value of the ClassExplorer is something that is missing from the other libraries and its utility has been proven extremely beneficial in practice. Hence, although MLC++ and WEKA are very good when one needs to have an organised base of **existing** ML algorithms they fall behind when one concentrates on the decision tree researcher/user.

The rest of this paper is organised in four sections. In the next section we present the basic characteristics of virtually any decision tree algorithm. Next, we elaborate on how to accommodate a mapping of such an algorithm to a flexible, components-based, object-oriented framework. We, then, present ClassExplorer and we conclude with intriguing topics of future research.

2. Decision Tree Building

The first step in building a decision tree learner is the acquisition of training data from some external source. Next, one has to pre-process (or filter) those data in order to bring them in an appropriate form for the concept builder. That step may include the discretisation of continuous features or the treatment of missing values. In a broader view, it may also contain the exclusion of some attributes, or the reordering of the instances inside the training instance-set.

Subsequently, the derived instance-set is used to build the decision tree. The basic framework of the top down induction of decision trees is rather simple. One has to

divide the instances at hand into two distinct sub-sets using a test value. This procedure continues for every newly created sub-set until a termination (or pre-pruning) criterion is fulfilled. The last optional step is that of pruning the obtained decision tree.

Finally, having a model of the training data one can use it to classify/predict unknown instances. Figure 1 illustrates the basic steps during decision tree building.

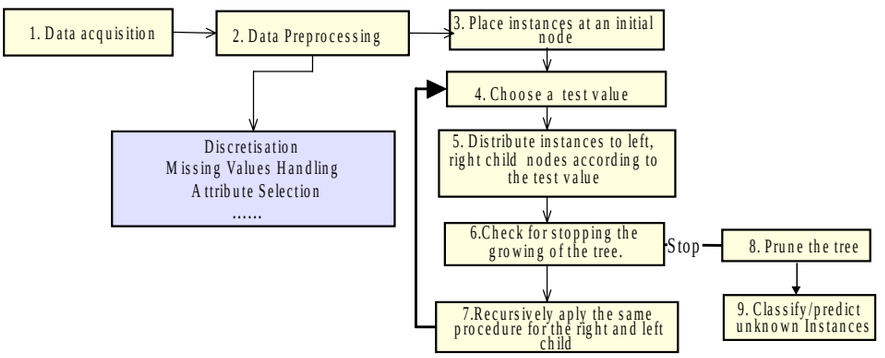

Figure 1. Basic steps during the decision tree creation/use.

Although the proposed scheme assumes a batch mode learner, it can easily be extended to incorporate incremental-mode builders.

3. Library Design

A basic requirement for the implementation of the library was the suitability of the development environment. A components library is best (if not only) suited under an object-oriented language. That is the case since inheritance and polymorphism, basic characteristics of object-oriented languages, are basic characteristics of a components library too. Another basic requirement was ease-of-use, which means the availability of integrated environment tools to boost productivity as well as capability of producing user-friendly graphical interfaces.A further desired characteristic of the development environment was that it should be a well-known and established tool with natural support for libraries and well suited to emerging technologies.

With these considerations we chose Java as the coding language. Java is fully object-oriented (in contrast with C++ where more conservative programming approaches can be used). Furthermore, Java is known to reduce production time and its “no pointers” approach makes programs less error-prone. Moreover, it has been designed with class libraries in mind, (since even the core of the language incorporates extended libraries of classes), which means that there is an extended build-in functionality regarding the organization of the libraries and their mutual collaboration. Finally, Java offers the ability of modified or customized versions of this software that could run over the Internet.

Having determined the programming environment, we designed the library to accurately fit both the object-oriented approach and the aforementioned basic decision tree algorithm. This means that:

- We identified the basic, relatively-stable parts during the decision tree lifecycle.
- We transformed them to primitive components (abstract classes) with specific minimal functionality.
- We used this minimal functionality to interconnect components into a central decision tree framework.

For example, the minimal needed functionality of a primitive component for the pre-processing step is one function that takes as input a set of instances and returns a modified set of instances. It is up to the user to build on this basic functionality and construct complicated preprocessing classes. There is no need for other components to know what happens inside a pre-processing class; they just use its result. In other words, the interconnection framework establishes a minimum level of collaboration and agreement between the components of the library.

A simple enumeration of the basic building blocks into the abstract classes framework and the role of each one on the building of decision trees is presented in Table 1.

Table 1: Basic Components of a Decision Tree algorithm.

Component	Role on decision trees framework
Parsing	Acquisition of data from various sources
Pre-Processing	Discretization, missing values handling etc.
Splitting	Select test-values to be used during the expansion of the decision tree
Distribution	Statistics about values' distribution among problem classes
Pre-Pruning	Stop the expansion of the decision tree
Post-Pruning	Prune the tree after it has been build
Tree Build	Batch mode, incremental mode builders
Classify	Classify instances using a decision tree

Although a full description of every structure and component of the library is beyond the scope of this paper, a simple example is necessary to point out some of the system characteristics. Suppose that the shapes at the left part of Figure 2 represent the basic abstract classes of the library and that arrows between them represent interconnections between those classes. For example, if shape 1 represents a *TreeBuild* class and shape 4 represents a *Splitting* class then there must be an interconnection between them since a tree builder has to decide about splitting values during the process of tree building. The same way, shapes 2 and 3 may represent a basic *Classifier* class (that uses the output of the tree builder) and a *Distribution* class (that is used by the *Splitting* Class) accordingly.

Now, suppose that the right part of Figure 2 represent classes that inherit from the abstract classes of the library. For example, the two produced shapes next to shape 1 may represent an incremental or batch mode builder. The same way, the two produced shapes from shape 4 may represent classes that use the information gain or the chi-square criterion to choose best splitting values. Since the basic interconnections between classes are hard coded inside the basic abstract classes, it is easy to use tree-builders and splitting-value-selectors at any combination. This architecture offers the researchers an easy way to focus on the points of interest during the decision tree's lifecycle. Furthermore, it gives them the ability to test their own extensions with a

variety of already built components and a selection of datasets ([9] is a possible source).

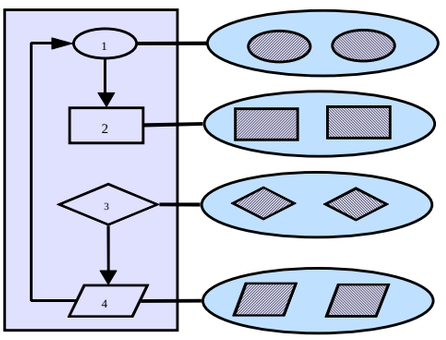

Figure 2. An example of library's architecture.

Of course, the library is accompanied with a general set of classes that represent data structures (like *Instance* or *Attribute*), containers for those structures to easily create data collections, as well as an extended set of methods to manipulate them. Furthermore, there is a general decision-tree wrapper, which helps combining the decision tree components to a decision tree entity. This wrapper contains methods to set the desired tree builder class, splitting class, pruning class, classification method etc., as well as methods to access the representative objects of those classes. Since this wrapper is aware of the components that comprise the decision tree entity, it internally handles the needed component interconnections. The library is also equipped with a number of tools (like timers or various decision tree printers) that provide additional functionality.

In its current state, the library is not directed to the implementation of standard decision tree algorithms (like C4.5) but rather to the implementation of the components that comprise those algorithms. It is up to the researcher to use already built components to simulate the behaviour of a desired algorithm. However, it is relative easy to create classes that resemble the behaviour of major current algorithms and developing a repertoire of them is a prominent topic in our agenda.

4. Class Explorer

A basic problem we confronted at the very early steps of the library's design was an easy way to accomplish basic object testing and algorithms' combinations. Although those problems could be tackled by writing and compiling new Java programs (that would act as wrappers), we found this approach rather cumbersome and thus unsuitable for research and development use.

This problem motivated the conception of the ClassExplorer, an interface tool for controlling and combining algorithms for decision trees. ClassExplorer is based on a simple interpreted programming language, which resembles Java and offers object construction "on the fly" from classes unknown during compilation time. This way we attain independence from the current state of the library while at the same time providing a flexible general tool for object manipulation. An interesting side effect of the independent nature of ClassExplorer is that, although it was developed as an

accompanying tool for the library, it can be used in a variety of tasks where interactive object construction and manipulation is required.

Some of ClassExplorer's capabilities are:

- It can instantiate different objects of previously unknown classes and map different variable names to different objects.
- It offers the capability of using/combining the methods of any instantiated object.
- It can make use of non-object primitive data types as *int* or *boolean.*
- It can organize actions into scripts and thus, provide additional functionality.
- It offers a set of special instructions that, for example, manipulate the instantiated objects or change the working directory of the system, or set paths where the system should search for classes.

On the other hand, the basic limitations (or, more accurately, as of yet missing features) of ClassExplorer are:

- There is no build-in program flow control.
- There is no support for mathematical operations.

One has to distinguish between the tasks that can be accomplished using the ClassExplorer and those that have to be done using pure Java. For example, ClassExplorer cannot create a new component of the library. On the other hand, it is naturally suited to tasks where one wants to conduct experiments with different combinations of components or wants to "take a look" at intermediate results of some algorithm's component. Furthermore, the capability of organizing commands to scripts makes it possible to organize test scenarios (like *k*-fold cross-validation[1]) over one or more datasets.

5. Conclusions and Future Directions

Due to space restrictions we cannot present a full example of the library's usage. The interested user can look at *htpp://www.cti.gr/RD3/Eng/Work/DTT.html* for the library, examples and further documentation.

We anticipate that researchers and specialist business consultants could be prime users of the toolkit (they will be able to use the interconnected framework to build over new or already implemented ideas without having to worry about essential but uninteresting parts of a decision tree learner). We also expect that students will greatly benefit from using the toolkit, as it constitutes an integrated environment for practicing machine learning and software engineering skills. The ClassExplorer in particular, should prove very helpful for all users, as its interactive nature provides insight into the components behavior.

It is relatively easy to wrap some of the library components and ClassExplorer so as to construct a user-friendly interface for non-specialist business users. This interface would construct commands for the ClassExplorer, which in turn would create and manipulate the required library objects. This is a three-layer architecture as illustrated in Figure 3, where four distinct (but coupled) steps implement a decision tree development lifecycle. This would enhance the proposed toolkit with a

[1] The Class Explorer already contains several scripts for the most common tasks like k-fold cross validations

functionality that has underlined the commercial success of existing software, like Clementine [10].

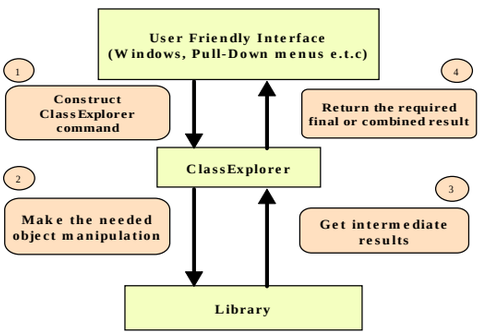

Figure 3. A 3 three-layer architecture for a business application over the library.

There is also a number of scheduled improvements that will provide added functionality and ease-of-use. We intend to incorporate a number of visual components to the library (with an interactive decision tree visualizer being the top priority). Furthermore, we intend to produce a graphical version of ClassExplorer, where every created object and its methods will be represented as visual components. The user will be able to drag-and-drop such components, visually interconnect them and simulate their combined behaviour. Finally, we intend to produce a client/server version of the library for use over Internet.

Whenever decision trees are the preferred means for knowledge representation the proposed library offers a serious alternative to other established machine-learning libraries. The reduced problem scope provides a solid and easy to familiarize-with framework; furthermore, the ClassExplorer, as a productivity tool, offers an interactive mediator between the user and the library, and component authors are assisted by the coding language (Java). Finally, the already built infrastructure provides the core in which (already scheduled) user-friendly access or networking capabilities can be seamlessly integrated.

References

[1] Papagelis, A., Drossos, N. (1999). *A Decision Trees Components Library*. Diploma Thesis at Computer Engineering and Informatics Department, University of Patras, Greece (in Greek).
[2] Kohavi, R., John, G., Long, R., Manley, D., Pfleger, K. (1994). MLC++: A Machine Learning Library in C++. *Proceedings of TAI'94*.
[3] Witten, I., Frank, E. (2000). *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann Publishers, San Mateo, CA.
[4] Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning* Vol. 1, No.1, pp. 81-106.
[5] Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann , San Mateo, CA.
[6] Aha, D., Kibler, D. (1991). Instance-based learning algorithms, *Machine Learning*, 6:37-66.
[7] George, H.J., Langley, P. (1995). Estimating Continuous Distributions in Bayesian Classifiers. *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pp. 338-345.
[8] Kohavi, R. (1995). The Power of Decision Tables. *Proceeding of European Conference on ML*.
[9] Blake, C., Keogh, E., Merz, J. (2000). *UCI Repository of machine learning databases*. Irvine, University of California, Dep. of Information and Computer Science.
[10] Clementine. http://www.spss.com/clementine/