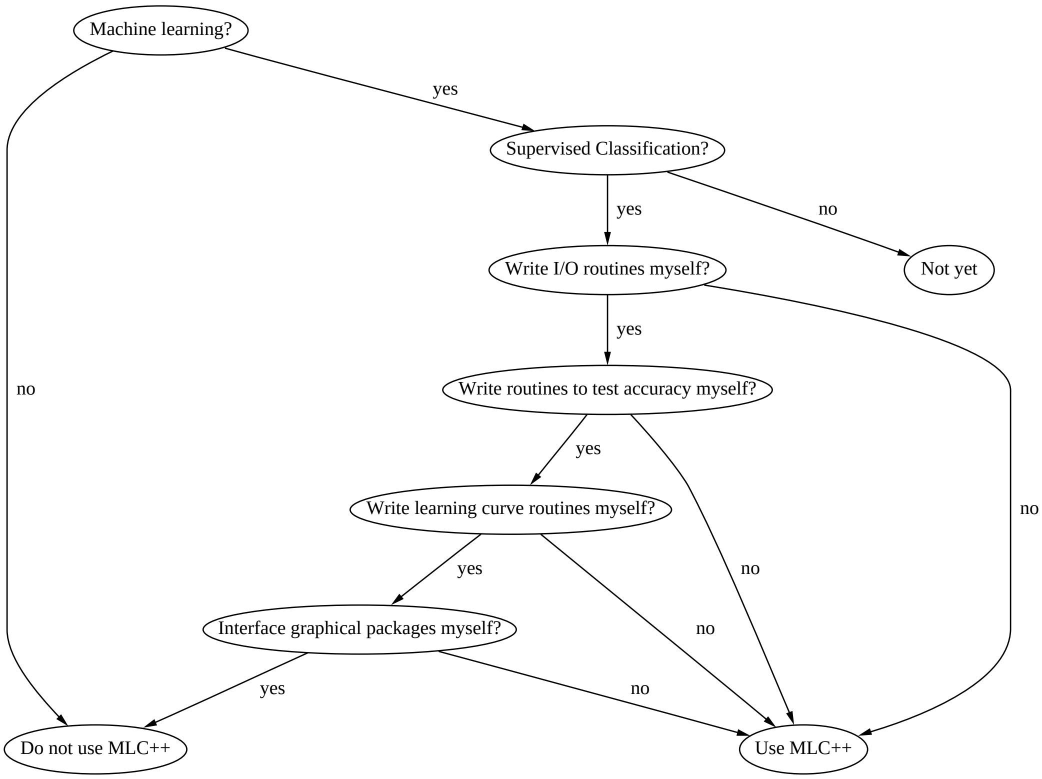
$\mathcal{MLC}$++ Tutorial

A Machine Learning library of C++ classes.

Ronny Kohavi
mlc@CS.Stanford.EDU

November 24, 1995



When to use $\mathcal{MLC}$++

Preface

$\mathcal{MLC}$++ is a Machine Learning library of C++ classes.[1] The current implementation supports supervised learning of concepts using decision trees, decision graphs, nearest-neighbor (instance-based), probabilities (Naive-Bayes), and more. In addition to providing algorithms, the library provides wrapper algorithms to Given an induction algorithm, wrappers produce learning curves, perform accuracy estimation, generate statistics, and attempt to improve performance by methods such as feature subset selection (John, Kohavi & Pfleger 1994). $\mathcal{MLC}$++ also provides tools to view the resulting structures and the instance space.

This tutorial will take you on a guided tour of the $\mathcal{MLC}$++ classes and conventions, making it easy to code similar machine learning algorithms or to modify existing ones. $\mathcal{MLC}$++ is aimed at researchers and students who want to modify algorithms and write algorithms, rather than simply execute a specific algorithm. Using $\mathcal{MLC}$++ requires good knowledge of C++, a good C++ compiler, and a fast machine.

$\mathcal{MLC}$++ is constantly being improved, both in the available features, and in quality. We are regularly testing it for illegal memory accesses and memory leaks using TestCenter. However, just like every program over a few hundred lines, it still has bugs. Please send any bug reports to `mlcpp-bugs@CS.Stanford.EDU`. Questions or requests for help should be addressed to `mlcpp-help@CS.Stanford.EDU`.

Acknowledgments The development of $\mathcal{MLC}$++ would not have been possible without the support of Nils Nilsson and Yoav Shoham. George John, Pat Langley, Richard Long, Ofer Matan, Karl Pfleger, Scott Roy, and Dan Sommerfield have been involved in design issues. Robert Allen, Brian Frasca, James Dougherty, Jamie Chia-Hsin Li, Richard Long, David Manley, Svetlozar Nestorov, Dan Sommerfield, and Yeogirl Yun have been actively involved in programming and design of $\mathcal{MLC}$++ classes.

[1]General information about the library can be obtained through the world wide web at URL `http://robotics.stanford.edu:/users/ronnyk/mlc.html`

Contents

1 Introduction

The $\mathcal{MLC}$++ library provides tools that can help students and researchers experiment with supervised learning algorithms.

1.1 Available Tools

The library currently includes the following tools:

Input/Output Read and write Quinlan's format (Quinlan 1993), which is a superset of Irvine's format (Murphy & Aha 1995). Quinlan's format defines a machine readable file that allows automatic parsing of the data file.

Accuracy estimation Besides holdout (train/test), which is the most common accuracy estimation method, $\mathcal{MLC}$++ supports cross-validation, stratified cross-validation, bootstrap .632, and generation of learning curves.

Graphical display of structures Interface routines are provided to AT&T's `dot` and `dotty` programs which display the resulting trees and graphs.

General logic diagrams (GLDs) Show general logic diagrams, which are projections of multidimensional discrete spaces on two dimensions. They are similar to Karnaugh maps, but are generalized to non Boolean input/output. GLDs are described in Michalski (1978), Wnek, Sarma, Wahab & Michalski (1990), and Thrun *et al.* (1991).

Top down induction of decision trees Induce decision trees top down using the ID3 algorithm (Quinlan 1986), or interface the commercially available C4.5 (Quinlan 1993).

Bottom up induction of OODGs Induce Oblivious read-Once Decision Graphs bottom up using the HOODG algorithm (Kohavi 1994).

Nearest neighbor An implementation of Aha's IB1 and IB2 (Aha 1992) with many extensions and variations.

Decision table An implementation of decision table induction algorithm, which is useful when couple with feature subset selection (Kohavi 1995).

1.2 Intended Audience

$\mathcal{MLC}$++ is first of all a C++ class library. Although it provides utilities and tools that can be used standalone[2], the main benefit from use of the library is the ability to piece together classes that the library provides, and instantiate them for exploring new algorithms or variants of existing algorithms.

The tutorial reviews some basic machine learning material, and some of the exercises can help in understanding fundamental machine learning concepts, but it is not intended to be used as a primary guide to machine learning.

The tutorial will not give enough information to allow writing your own algorithms. It is a basic introduction to introduce the important concepts and to show some of the features of $\mathcal{MLC}$++.

This tutorial assumes that

[2] see the utils.ps document

- You are familiar with the C++ language, and have access to AT&T's C++ compiler release 3.0 or higher.
- You are familiar with basic Unix, including the ability to edit files.
- You are working under Unix `csh` or `tcsh`. There are a few scripts that are used for convenience, but they are not necessary to work with $\mathcal{MLC}$++.
- You have a machine running the X Window System (if you wish to use the graphical capabilities of $\mathcal{MLC}$++).
- You have familiarity with basic machine learning concepts, or you are studying machine learning in parallel with this tutorial.

1.3 Conventions and General Comments

The tutorial follows the convention of showing code, code output, and C++ class names in typewriter style.

$\mathcal{MLC}$++ source files have a lowercase ".c" suffix (see coding standards document for reasons). Users working with emacs may want to change some defaults to our suggested emacs defaults by adding `(load "<mlcdir>/emacsMLC.el")` to their `.emacs` (replace `<mlcdir>` appropriately).

The exercises at the end of sections add information, and are intended to be done as part of the tutorial. For each exercise, a number denoting the "relative difficulty" is given in parentheses. Exercises may require you to modify short portions of programs; programs are all available in the tutorial directory `$MLCDIR/tutorial`. Exercises which are meant to aid students in understanding machine-learning concepts, and which are not necessary for researchers who are interested in the library itself, are marked with "Students".

1.4 Related Documentation

While not required for this tutorial, the following documents are currently available for readers who wish to understand $\mathcal{MLC}$++ better:

util The utilities provided by $\mathcal{MLC}$++. We suggest reading the util document before going through this tutorial.

cc-coding C++ coding Standards in $\mathcal{MLC}$++. A description of the coding standards for writing C++ code in $\mathcal{MLC}$++. The description here can be used as a general guideline for programming in C++, independent of $\mathcal{MLC}$++. It is a low-level guide that does not discuss important issues of design.

mlc-coding $\mathcal{MLC}$++ coding standards. Coding standards and conventions used in $\mathcal{MLC}$++.

environ The $\mathcal{MLC}$++ environment. The environment, directory structure, and useful scripts. This document is mostly intended for developers of $\mathcal{MLC}$++, but you may find it useful to browse through it.

GLD-status Description of the implementation and current status of General Logic Diagrams. The document GLD-design describes the design considerations and extensions.

2 Supervised Classification Learning

This section describes the basic machine learning terminology required to understand the tutorial. We begin with an informal description, which is followed by the formal definitions.

2.1 An Informal Introduction to the Nomenclature

Xema is a company that receives loan applications and has to approve or reject each one. Xema has been doing credit approval for years, and has human experts that evaluate application forms. The company's CEO wants to replace some of the experts with a program that will help process applications faster and will require less manpower.

In machine learning terms, the CEO is looking for a *categorizer*[3], which will label each application as either *reject* or *accept.* The *induction task* is to come up with such a categorizer. The task is inductive (as opposed to deductive), because there will be no formal justification (*i.e.*, a proof) of the resulting categorizer. For many real domains, it is simply impossible to *prove* correctness of any rule. We will be satisfied with a system that performs well most of the time.

In order to induce such a categorizer, the *induction algorithm* must have some input. We will deal with induction algorithms that accept raw data as input. This type of supervised learning is sometimes called *tabula rasa* learning, as no domain knowledge is given to the learning algorithm. In our example, the input will be the set of all previous credit approval applications. Each credit approval application is an *instance.* In order to *train* the algorithm, we will label each instance with the category we would want the categorizer to categorize such an instance *i.e.*, accept or reject.

Instances are described as a list of *attribute* values. This is similar to a relational database, or flat-file. All information about an instance must be expressible as a fixed list of values. In our example, the attributes might include age, sex, marital status, social security number, type of car, *etc.*

The set of labelled instances that we give the induction algorithm is called a *training set.* Given a training set, we can run the induction algorithm, which generates a categorizer. A categorizer can have many forms, but usually it will be some *structure* and an *interpreter.* The interpreter tells us how to use the structure to categorize, or label the unlabelled instances. Examples of categorizers are:

Table The structure is a table of labelled instances. The interpreter takes an unlabelled instance and searches for it in the table. If it is found, the category returned is the label associated with the instance in the table. If the category is not found, the interpreter returns *unknown.*

Note that we have not specified how to induce the table. The table may simply be all instances in the training set, or it may include unseen instances, or it may include the set of all possible instances.

Nearest neighbor The structure is a table of labelled instances. The interpreter takes an unlabelled instance and searches for the instance in the table that has the closest distance (under some measure of distance). The categorizer returns the label of the instance found.

Note that under this approach, the categorizer will never returned *unknown* (assuming the table is not empty). This may not be appropriate in some domains. For example, in our

[3] In $\mathcal{MLC}$++ we use the term categorizer instead of classifier to avoid conflicts with C++ classes.

loan approval example, it may be better to say unknown on an uncommon application, and let the human experts look at the application.

Decision Tree The structure is a tree. Each interior tree node contains a test; the edges from each node are labelled so that they cover all possible test results, *i.e.*, the edge labels are mutually exclusive and exhaustive. The leaves contain categories. The interpreter starts at the root, evaluates the test at the node, and branches according to the result. This process is repeated until a leaf is reached. The categorizer returns the category at the leaf. Figure 5 on page 12 on page 12 shows a decision tree.

Boolean Formula This type of categorizer is valid only for Boolean domains, *i.e.*, both the attributes and the label are Boolean. The structure is a formula such as $A \wedge (B \vee C)$, where A, B, and C are attributes. The interpreter evaluates the formula according to the usual rules for logical operators, and categorizers the instance as the resulting value.

Inducing a categorizer is not the end of the story. It is extremely important to estimate how good the induced categorizer is. One possibility is to show the categorizer's structure to a domain expert, *i.e.*, someone who understands the domain well, and ask for his opinion. Another possibility is to test the categorizer on a test set, *i.e.*, new instances that were not used during training, and evaluate its performance. It is usually assumed that the training set and test set are random samples (i.i.d. samples) from some underlying distribution.

A *test set* is a set of instances that were not used to train the algorithm. A test set is used to predict the accuracy of the algorithm on future instances. Xema would be wise to set aside some fraction of its past applications and test the resulting categorizer on those instances.

2.2 Formal Definitions

An *attribute*, or *feature*, is a variable that takes a value from a predefined domain. Examples of attributes are: Sex (male or female), color (red, green, or blue), and temperature (real number).

An *instance* is a list of attribute values. An instance is associated with an *attribute schema*, which defines the names and domains of the attributes. A *labelled instance* is an instance augmented with a special attribute, called *label*, or *category*.

A *dataset* is a set of labelled instances, all associated with the same attribute schema. Readers familiar with databases may note that a dataset corresponds to a relational database (cf. Ullman (1988)). A *training set* is a dataset on which a learning algorithm is *trained.* A *test set* is a dataset on which a learning algorithm is *tested.*

A *categorizer* maps an unlabelled instance to a label. A *supervised learning algorithm induces* a categorizer from a training set.

3 Getting Started

To setup $\mathcal{MLC}$++, you need to know the "root," *i.e.*, where the library resides. Please refer to your local guide on the location of the root directory, which will be referrred to as `$MLCDIR`.

The file `setup` sets up the environment variables that are needed, and the file `aliases` sets up useful aliases. To start working with $\mathcal{MLC}$++, you should type

```
source $MLCDIR/setup
source $MLCDIR/aliases
```

You may want to run the setup script from your .login and the aliases script from your .cshrc by adding the above lines to those files. The scripts will redefine the `MLCC` command which should be used exactly as if it were the compiler (*e.g.*, as if it were `CC`). This alias defines the appropriate include directories and link libraries.
To test that your setup is correct, type

```
cp $MLCDIR/tutorial/setupTest.c .
MLCC setupTest.c
a.out
```

If everything compiles correctly, you should see the following when executing a.out

```
setupTest executing...
setupTest OK
```

If something does not work properly, see the troubleshooting guide (Appendix A on page 25).

4 Building an Induction Algorithm

In this section, you will write a program that runs the ID3 induction algorithm (Quinlan 1986) on a training set. You will then display the induced tree, and finally, you will test the tree on a test set. No specific knowledge of the algorithm is needed to follow the tutorial.

ID3 is an algorithm for top-down induction of decision trees (TDIDT). Given a dataset, it induces a decision tree which can be used to categorize instances.

4.1 Inducing a Decision Tree

Figure 1 on the following page shows a simple program that runs ID3 on a small dataset called "tutorial," and gives an ASCII description of the resulting tree as shown in figure 2 on the next page.

The program begins with two includes. The first include (basics.h) is mandatory in any program using $\mathcal{MLC}$++. The second include is required for using the ID3 induction algorithm. Line 9 declares an object "inducer" of type `ID3Inducer`. The argument "Xema" is simply a description which is ignored, except for display purposes. Line 11 then calls `read_data()` to read the training set. This call assumes that the attribute schema file has a ".names" suffix, and that the data file has a ".data" suffix. File names are searched according to the directories listed in `$MLCPATH`. The default setup searches in the current directory, then in `$MLCDIR/db`. Line 13 produces the categorizer by "training" on the training set. Finally, line 14 displays the induced categorizer structure.

Looking at the output (Figure 2 on the following page), we can see that the root test is "car", and there are three branches, "cheap" goes to node 1, "average" goes to node 2, and "expensive" goes to node 3. Nodes 1 and 2 are labelled by reject. Node 3 executes another on "hair color." Everything will reach accept, except hair color brown which will reach reject.

4.2 $\mathcal{MLC}$++ Streams: Improving the Output

It is not always easy to see the decision tree from the ASCII output. $\mathcal{MLC}$++ allows generating X windows output by interfacing the `dot` and `dotty` program to display trees and graphs (Koutsofios & North 1994). In this section, we will look at $\mathcal{MLC}$++ streams.

```
// Tutorial program 1
// Run the ID3 algorithm on a test set and display the tree in ASCII.

#include <basics.h>     // Must be included first in any program using MLC++
#include <ID3Inducer.h> // Include ID3Inducer class declarations

main()
{
   ID3Inducer inducer("Xema");      // "inducer" is an object of ID3Inducer.
                                    // The argument is the "description."
   inducer.set_unknown_edges(FALSE);// Avoid edges for "unknown" attributes.
   inducer.read_data("tutorial");   // Read data files tutorial.names and
                                    //   tutorial.data
   inducer.train();                 // Produce categorizer
   inducer.display_struct();        // Display structure of categorizer

   return 0;                        // Return success to shell
}
```

Figure 1: The program “dt1.c” to induce and display a decision tree using ID3.

```
MLC++ Debug level is 1, log level is 1
OPTION PROMPTLEVEL = required-only
OPTION REMOVE_UNKNOWN_INST = false
OPTION CORRUPT_UNKNOWN_RATE = 0
Reading tutorial.data done.
Tree has 8 nodes, and 6 leaves.
Rooted Decision Graph Categorizer Xema
Root: car

[0](car) : [0]--(cheap)-->[1][0]--(average)-->[2][0]--(expensive)-->[3]
[1](reject) :
[2](reject) :
[3](hair color) : [3]--(black)-->[4][3]--(brown)-->[5][3]--(blonde)-->[6][3]--(grey)-->[7]
[4](accept) :
[5](reject) :
[6](accept) :
[7](accept) :
```

Figure 2: The output of “dt1.c’

```
// Tutorial program: output to a file

#include <basics.h>
#include <ID3Inducer.h>

const MString treeFile("tree.txt"); // Use MLC++ string class.

main()
{
   Mcout << "The tree will be stored in " << treeFile << endl;
   ID3Inducer inducer("Xema");
   inducer.set_unknown_edges(FALSE);
   inducer.read_data("tutorial");
   inducer.train();

   MLCOStream outfile(treeFile);
   outfile << "The tree produced by ID3 on the tutorial dataset:" << endl;
   inducer.display_struct(outfile);// Write structure to the file
                                   // instead of the standard output.

   return 0;
}
```

Figure 3: The program "dt2.c" to demonstrate $\mathcal{MLC}$++ I/O operations.

$\mathcal{MLC}$++ provides extended streams for input/output. These streams are "safe," *i.e.*, operations are checked for errors, and they provide conveniences such as automatic wrapping of words. A stream can also be a special X-stream, which functions will recognize and start an X-window for.

File streams

The simplest streams are file streams, and they behave in almost the same way as C++ streams, except that automatic wrapping can be set (and it is set for the default output stream). To declare an output stream, use the class `MLCOStream`; for declaring an input stream, use the class `MLCIStream`. Both are initialized with a file name, and close the file when the class destructor is invoked. $\mathcal{MLC}$++ streams provide almost all common operations on C++ streams. If you need to perform a rare operation, `get_stream()` will return you a regular C++ stream from an $\mathcal{MLC}$++ stream.

The streams `Mcout`, `Mcin`, and `Mcerr` are $\mathcal{MLC}$++ streams that corresponds to the usual `cout`, `cin`, and `cerr` C++ streams. Mcout is automatically wrapped according to the width declared in `LINE_WIDTH` (default to 79). Most operations that require an $\mathcal{MLC}$++ stream default to `Mcout`.

Figure 3 shows how to send output to the standard output (Mcout), and how to output the tree structure into the file "tree.txt." Note that `display_struct()` has the default stream `Mcout` which was used in the previous program.

Display Preferences

When the structure of the tree was displayed in the previous programs, the resulting display was in ASCII. For some categorizers, including decision trees and graphs, $\mathcal{MLC}$++ supports other display preferences. $\mathcal{MLC}$++ currently provides three preference classes:

```
// Tutorial program: preferences

#include <basics.h>
#include <MLCStream.h>
#include <ID3Inducer.h>

main()
{
   ID3Inducer inducer("Xema");
   inducer.set_unknown_edges(FALSE);
   inducer.read_data("tutorial");
   inducer.train();

   MLCOStream psOut("dt3.ps");
   DotPostscriptPref psPref;              // Declare Dot preference
   // Default orientation is LandScape.  Change to portrait
   psPref.set_orientation(DotPostscriptPref::DisplayPortrait);
   inducer.display_struct(psOut,psPref);    // Display to file using given
                                          //    preferences

   MLCOStream dotOut("dt3.dot");          // Now generate dot output
   DotGraphPref dotPref;
   inducer.display_struct(dotOut, dotPref);

   return 0;
}
```

Figure 4: The program "dt3.c" that uses display preferences.

ASCIIPref The default for most operations. There are no options for this preference.

DotGraphPref Generate `dot` (Koutsofios & North 1994) format. The output can then be given to `dot` or `dotty` which can generate postscript, or display the graph on an X-terminal. There are no options for this preference.

DotPostscriptPref Generate postscript output directly using dot. Options that may be set are: page size, graph size, orientation, and fill ratio (see the dot manual for a description of these options).

Figure 4 is an example of how to use these options. Note that the member function `display_struct()` has a second argument which defaults to ASCIIPref, and that default was used in the previous examples. The program generates two files: dt3.ps (shown in Figure 5 on the next page) and dt3.dot. The first can be printed or viewed through a postscript display program such as ghostview; the latter can be converted to postscript using the `dot` program, or can be viewed on an a device supporting X-windows by typing "`dotty dt3.dot`" and using the right-mouse button on an empty part of the canvas to quit.

XStreams

This part of the tutorial must be executed on a device supporting X-windows.

As an alternative to generating postscript of dot output and running dotty, $\mathcal{MLC}$++ provides a direct interface to X-windows. XStreams are nameless streams which are used for graphical displays.

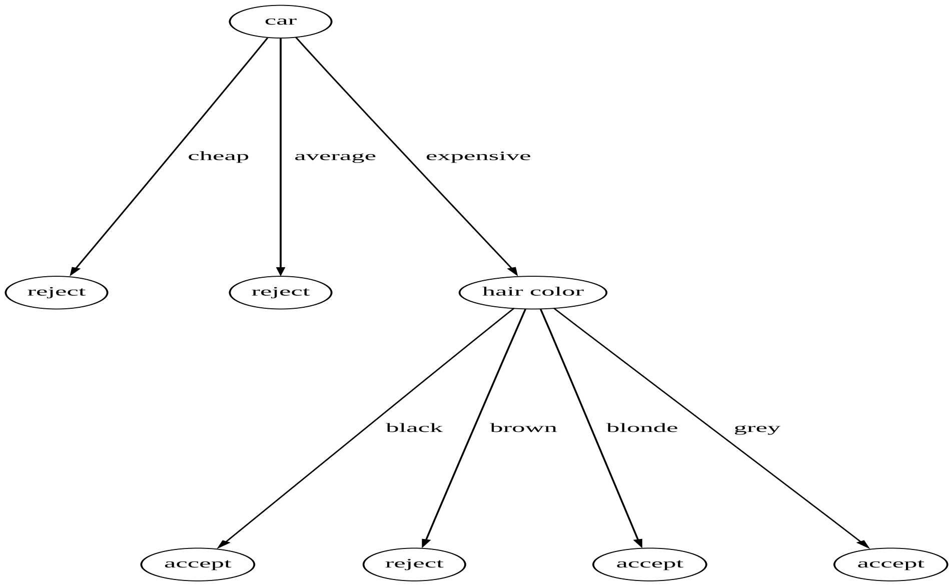


Figure 5: The generated postscript file "dt3.ps."

The function `display_struct()` supports XStreams under some categorizers and inducers, including decision trees and graphs. An XStream behaves just like a file, but only specific preference which make sense are supported.

Figure 6 on the following page shows how to open an XStream and send the display to the stream. When the program is run, the graph is displayed and the program waits until the X-window showing the graph is closed by clicking the right mouse button and choosing "quit."

4.3 Following the Execution

In many cases, it is helpful to see what the algorithms are doing. This can help in understanding why the algorithm produces the result it does, and it can also help debugging code.

Many classes in $\mathcal{MLC}$++ have options which can be set. An important option is the log-level. A log-level higher than zero causes display of log information, with higher log-levels usually producing more output.

Figure 7 on the next page shows an example of setting the log-level to 2, and Figure 8 on page 14 shows the output. A log-level of 2 causes the ID3 algorithm to display the mutual-information (or information gain) at each variable tested. The mutual information indicates how good the split is estimated to be. The higher the value, the better the split is.

Running the program shows that at the root, there is a clear advantage to choosing "car," and at the second level there is an advantage to choosing "hair color" over "own house."

The default log-level for all classes that have a log-level option is taken from the environment variable `LOGLEVEL`. If the environment variable is undefined, the default log level is zero. This allows changing the log-level without recompiling. Note, however, that a run-time setting overrides the default behavior. Also be aware that in some cases, setting the log-level for the whole program may produce very large amounts of output.

For example, executing the following command from csh will cause future runs to default to log-level of 2:

```
// Tutorial program.  Output to XStream

#include <basics.h>
#include <MLCStream.h>
#include <ID3Inducer.h>

main()
{
   ID3Inducer inducer("Xema");
   inducer.set_unknown_edges(FALSE);
   inducer.read_data("tutorial");
   inducer.train();

   MLCOStream out(XStream);             // Declare out to be XStream
   DotGraphPref dotPref;
   inducer.display_struct(out,dotPref);// Display the structure to the XStream.

   return 0;
}
```

Figure 6: The program "dt4.c" that uses XStreams

```
// Tutorial program: log-level

#include <basics.h>
#include <ID3Inducer.h>

main()
{
   ID3Inducer inducer("Xema");
   inducer.set_unknown_edges(FALSE);
   inducer.read_data("tutorial");

   inducer.set_log_level(2); // set log-level to 2

   inducer.train();

   return 0;
}
```

Figure 7: The program "dt5.c" that sets the log-level.

```
MLC++ Debug level is 1, log level is 1
OPTION PROMPTLEVEL = required-only
OPTION REMOVE_UNKNOWN_INST = false
OPTION CORRUPT_UNKNOWN_RATE = 0
Reading tutorial.data done.
Testing attribute 0 (hair color): Mutual info is 0.0477309
Testing attribute 1 (car): Mutual info is 0.384859
Testing attribute 2 (own house): Mutual info is 0.0464393
Creating split on attribute 1 (car) at level 0
Testing attribute 0 (hair color): Mutual info is 0.360964
Testing attribute 1 (car): No reasonable split
Testing attribute 2 (own house): Mutual info is 0.321928
Creating split on attribute 0 (hair color) at level 1
Tree has 8 nodes, and 6 leaves.
```

Figure 8: The output of program "dt5.c"

```
setenv LOGLEVEL 2
```

You may wish to recompile the program in Figure 7 on the page before without the statement setting the log-level, and modify the environment variable. Most users generally like to work at LOGLEVEL 1 to see what options are available and because it shows you when something is happening.

If you would like to see which routine is printing the message (so you can understand where things are done), you can set the `SHOW_LOG_ORIGIN` environment variable to yes. Logging messages will then be preceded by the file and line number where they appear.

The rest of the tutorial is assumed to be run at `LOGLEVEL 0` and without showing the log origin. If you have set the above variables, please type

```
unsetenv LOGLEVEL
unsetenv SHOW_LOG_ORIGIN
```

4.4 Testing the Induced Categorizer

After an induction algorithm is run, we would like to evaluate its performance on unseen data. The `CatTestResult` class provides the ability to test a categorizer on a test set.

Figure 9 on the following page shows how to evaluate the categorizer's performance on the test set. Figure 10 on the next page shows the output. The generalization accuracy indicates the accuracy on unseen instances and the memorization accuracy indicates the accuracy on instances in the test set which were also in the training set. The accuracy is followed by $\pm$ and the theoretical standard deviation, and the range afterwards is the 95% confidence interval. Note that although the test set accuracy is 80%, all we can say about the true accuracy is that it will be in the range 54.81% and 92.95% with high probability. The width of the confidence interval is determined by the the size of the test set. With a test set of the size used—15 instances—we cannot determine the accuracy precisely. In real problems, one would like to test on at least a few hundred instances to ensure a smaller confidence interval. Exercise 0 deals with one possible way of improving the accuracy using a domain expert.

```
// Tutorial program : Run and test ID3

#include <basics.h>
#include <ID3Inducer.h>
#include <CatTestResult.h>  // required for CatTestResult

main()
{
   ID3Inducer inducer("Xema");
   inducer.read_data("tutorial");
   inducer.train();

   // CatTestResult requires a categorizer, the training set, and the
   //    test set.  The default extension for test files is ".test"
   CatTestResult result(inducer.get_categorizer(),
                        inducer.instance_bag(), "tutorial");
   Mcout << result << endl;  // Give statistics.

   return 0;
}
```

Figure 9: The program "dttest.c" that tests the categorizer on a test set.

```
MLC++ Debug level is 1, log level is 1
OPTION PROMPTLEVEL = required-only
OPTION REMOVE_UNKNOWN_INST = false
OPTION CORRUPT_UNKNOWN_RATE = 0
Reading tutorial.data done.
Tree has 10 nodes, and 8 leaves.
Reading tutorial.test done.
OPTION DISP_CONFUSION_MAT = false
OPTION DISP_MISCLASS = false
Number of training instances: 10
Number of test instances: 15.  Unseen: 14,  seen 1.
Number correct: 12.  Number incorrect: 3
Generalization accuracy: 78.57%.  Memorization accuracy: 100.00%
Accuracy: 80.00% +- 10.69% [54.81% - 92.95%]
```

Figure 10: The output of "dttest.c"

4.5 Exercises

1. [1] A simple inducer class called `ConstInducer` induces a category which is then returned by the `ConstCategorizer` (independent of its input). The category is simply the majority category in the training set. Modify the program "dttest.c" so that it uses a `ConstInducer` instead of `ID3Inducer`. Note that you will have to include `ConstInducer.h` instead of `Id3Inducer.h`. Add the statement from "dt1.c" to display the structure. What is the accuracy? What is the most frequent category?

   Note that in many real-world problems, the majority category may give well over 50% accuracy. For example, given a random population of humans that have to be categorized as HIV positive or negative, a ConstInducer that returns negative will do very well. A good induction algorithm should obviously do better than a constant inducer.

   Sometimes it is useful to view the confusion matrix, a matrix that shows what types of errors the algorithm is doing. For example, in a credit card company, it is less of a problem if a good applicant is rejected, but it may cost the company a lot of money of a bad applicant, who cannot pay the money he owes, is accepted. The confusion matrix shows how each example from the test set is classified and how it should have been classified. By typing `setenv DISP_CONFUSION_MAT yes` and running your program again, you can see the confusion matrix. Unless you want to see the matrix again, you should `unsetenv DISP_CONFUSION_MAT`.

2. [2] $\mathcal{MLC}$++ contains a string class called `MString`, which provides many constructors, including a constructor from `(char *)`. Operations like equality (`operator==`), assignment (`operator=`), and automatic conversion to `(const char *)` are provided.

   Modify the program "dttest.c" so that it uses a variable `dataFile` of type MString to represent the file name (the ID3 inducer should be used). Initialize `dataFile` with the constant "tutorial" as the filestem for the names/data files, and eliminate the multiple uses of the constant "tutorial" by replacing it with the variable.

3. [3] Many times you would like to give arguments to the program such that recompilation will not be necessary.

   a. $\mathcal{MLC}$++ provides functions to get options. These can either be set by environment variables, and, if unset, the user will be prompted. The function

      ```
      MString get_option_string(const MString& optionName,
                                const MString& defaultVal,
                                const MString& optionHelp)
      ```

      returns an MString with the option value. If the default is empty, the user will be prompted. If it is not, the user will be prompted only if the PROMPTLEVEL is basic or all (the default is required-only).

      To use this function, you must add the following to your program:

      ```
      #include <GetOption.h>
      ```

      At the appropriate place in your code, add the following:

```
MString dataFileHelp("Type the name of a file stem");
MString dataFile = get_option_string("DATAFILE", "", dataFileHelp);
```

Modify the program in Exercise 2, so that the `datafile` is initialized with the value of the option `DATAFILE`. What is the accuracy of the ID3-generated decision tree on the datasets "monk1-full" and "breast"?

b. A function to get a Boolean option is

```
Bool get_option_bool(const MString& optionName, Bool defaultValue,
                     const MString& optionHelp = emptyString)
```

Note that the second argument is a Boolean and you should pass either `TRUE` or `FALSE` as a default (the default is required here). Add code to your program that displays the induced tree on an X-terminal (using XStreams) if the option "DISPGRAPH" is yes (or true). If the option is not set, your program should not display the graph. (Hint: what is the default we want you to give the `get_option_bool` function.)

5 Data File Formats

This section describes the dataset file formats that $\mathcal{MLC}$++ recognizes by default. The file formats are based on Quinlan's C4.5 format (Quinlan 1993), which share the same *data* file format with the Irvine database (Murphy & Aha 1995). Note that most Irvine datasets do not include a *names* file that defines the dataset schema (*i.e.*, the attribute names and domains).

The *data* file and *test* files contain labelled instances, one instance per line. The *names* file defines the schema that allows parsing the data file. $\mathcal{MLC}$++ functions that read datasets require only a *filestem* and append the appropriate suffix of ".data," ".test," or ".names" corresponding to the data, test, and schema files.

5.1 The Names File

The names file describes the dataset schema. It describes the name and domain for each attribute and for the label. The default extension of the names file is ".names."

The vertical bar (|) starts a comment. Everything to the right of the vertical bar is ignored. Blank lines, spaces, and tabs may be used freely to increase readability. When they appear as part of a name, multiple white spaces (spaces, tabs) are compressed into a single space.

A names file starts with a label definition followed by attribute definitions. A *label definition* is a period terminated domain definition, and an *attribute definition* consists of an attribute name, followed by a colon, a domain definition, and terminated with a period.

A *domain definition* is either the reserved word "continuous" or a list of comma-separated domain values. *Domain values* and attribute names are strings that do not contain the characters comma (,), colon (:), backslash (\), vertical bar (|), or question mark (?)[4] Note that spaces are valid characters in strings, but multiple white spaces are compressed into a single space.

Figure 11 on the following page shows an example of the names file for the tutorial dataset, and Figure 12 on the next page shows the matching data file.

[4]These characters are actually allowed under special conditions.

```
| Names file used in the MLC++ tutorial
| Ronny Kohavi

accept, reject

hair color: black, brown, blonde, grey.
car : cheap, average, expensive.
own house: yes, no.
```

Figure 11: The tutorial.names file

```
| tutorial data
| Ronny Kohavi

black, cheap,    yes, reject.
black, cheap,    no,  reject.
grey,  average,  no,  reject.
blonde,cheap,    yes, reject.
grey,  expensive,yes, accept.
blonde,expensive,yes, accept.
black, expensive,yes, accept.
brown, expensive,no,  reject.
blonde,average,  yes, reject.
| blonde,expensive,no,  reject.
```

Figure 12: The tutorial.data file

5.2 The Data File

The data file describes the dataset instances that are used to train the algorithm. The data in the file is usually called the *training set* (see Section 2.2 on page 7). The default extension of the data file is ".data."

The vertical bar (|) starts a comment. Everything to the right of the vertical bar is ignored. Blank lines, spaces, and tabs may be used freely to increase readability.

Each line in the data file describes one instance. Each instance is composed of comma-separated attribute values, and ends with a period. The attribute values must be from the domains of the corresponding attributes in the names file, and the whole instance must be one physical line in the file (*i.e.*, no newlines are allowed).

5.3 The Test File

The test file describes the dataset instances that are used to test the performance of the algorithm. The data in the file is usually called the *test set* (see Section 2.2 on page 7). The default extension of the test file is ".test." The format of the test file is the same as that of the data file.

5.4 Exercises

4. [3] The tree shown in Figure 5 on page 12 was shown to a *domain expert, i.e.*, someone who understand the domain on which the algorithm was run. The expert said that the tree does

not make sense to her. After trying to convince the expert to be more explicit, she marked the "accept" label connected by the edge "blonde" to the "hair color" node, and said that she would reject such application forms if the applicant did not own a house.
Copy the tutorial.data file to your directory, and try the following modifications:[5]

a. Add an instance corresponding to the new information from the expert, *i.e.*, a rejection for a blonde with an expensive car that does not own a house. Note that there is an instance exactly like the one you are adding that is labelled accept. By adding another instance with a conflicting label, the split on hair color will not be as good, and ID3 might change the tree built. Run ID3 and report the new accuracy and resulting tree.

b. When the expert learned that there was actually a case where someone was approved of a loan without owning a house, she requested that the instance be removed from the training set and suggested firing the idiot who approved that loan in the first place. Remove the instance (both with accept and reject labels). Run ID3 and report the new accuracy and resulting tree.

5. [7] Design a dataset for the target concept f over four Boolean attributes A, B, C, D:

$$f = B \vee C$$

that is, the Boolean label is one (true) if and only if the attribute B is one (true), or C is one (true).

a. Write a names file, a data file, and a test file. The test file should contain all 16 possible instances labelled according to the above target concept. The data file should contain five instances of your choice (labelled correctly). Show the dataset, derived tree, and accuracy.

b. (Students[6]) Suppose you were trying to "teach" the algorithm the target concept. Find a minimal set of instances such that ID3 will learn the correct concept. Do not change the names file.

   Hint: The number of leaves is a lower bound on the number of instances needed, since each leaf must contain at least one instance.

c. (Students) How many leaves are there in parity over n attributes $X_1, \ldots, X_n$. *Parity* is defined as

$$f = X_1 \oplus X_2 \cdots \oplus X_n \quad (\oplus \text{ denotes exclusive or})$$

   Explain your answer.

   Hint: can two instances be in the same node?

   One may conclude from this that parity-like concepts are very hard for decision trees to learn. Such concepts are rare in the real world.

[5]Note that since `MLCPATH` is set to search in the current directory before searching in the database directory, having tutorial.data in the current directory suffices for this exercise.

[6]Exercises marked with "students" do not introduce any new feature, but are aimed in helping students understand more issues in machine learning.

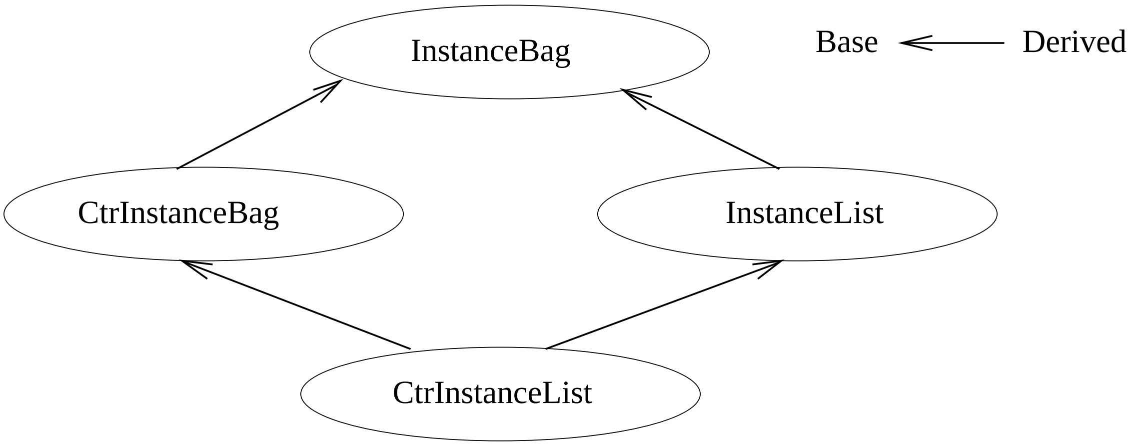


Figure 13: The Bag-List inheritance hierarchy.

6. [3] (Students) Design a data set for the target concept

$$f = (A \wedge B) \vee (C \wedge D)$$

over attributes A, B, C, D.

Train ID3 on the whole space (16 instances), and print the resulting graph. Circle the two isomorphic trees with 5 nodes. This problem is called the "replication problem." It can be solved by doing "constructive induction" (described below).

7. [2] (Students) Add two attributes to the above dataset. The first is called AB and will be one (true) iff both A and B are one. The second is called CD and will be one iff both C and D are one (there will still be only 16 instances in the space). Run ID3 and print the tree. This helped ID3 generate a small tree. One of the biggest problems is the construction of "useful" attributes (constructive induction). Note that ID3 had to pick AB and CD from all six attributes.

6 Bags and Lists

Datasets are represented in $\mathcal{MLC}$++ as either bags or lists of instances. The main difference between a bag and a list is that the latter is ordered whereas the former is not. Both may contain duplicate instances.

Some algorithms such as ID3 need to keep track of counters for efficiency reasons. Counters allow an algorithm to efficiently query the number of instances that have a given label, a given attribute value, or both a given label and a given attribute value. Since this functionality is required for both lists and bags, $\mathcal{MLC}$++ provides all four possible classes, arranged in inheritance diamond shown in figure 13.

A dataset can be read into a `InstanceList` (instance list) or a `CtrInstanceList` (counter instance list), and then passed as a bag to algorithms that require a bag (polymorphism). Algorithms such as ID3, which require a `CtrInstanceBag` (such as ID3), can only be passed the appropriate object, or an object of a derived class, such as `CtrInstanceList`.

Figure 14 on the following page shows a program that declares a `InstanceList`, causing the dataset to be read into memory, and then prints it out in two different ways. Line 12 simply directs the `InstanceList` to an output stream. Many $\mathcal{MLC}$++ classes have this ability

```
// Tutorial program: read and print dataset

#include <basics.h>
#include <InstList.h>  // Required for InstanceList

main()
{
   // The default extensions are ".names" and ".data."  They may be
   //   given as the second and third arguments.
   InstanceList list("tutorial");

   Mcout << "Easy printing:\n" << list << endl;

   Mcout << "Same display but manually:" << endl;
   for (Pix pix = list.first(); pix; list.next(pix)) {
      const InstanceRC& instance = list.get_instance(pix);
      Mcout << instance;
   }

   return 0;
}
```

Figure 14: The program "readil.c" for reading and printing the data in a dataset.

```
MLC++ Debug level is 1, log level is 1
OPTION PROMPTLEVEL = required-only
OPTION REMOVE_UNKNOWN_INST = false
OPTION CORRUPT_UNKNOWN_RATE = 0
Reading tutorial.data done.
Easy printing:
black, cheap, yes, reject.
black, cheap, no, reject.
grey, average, no, reject.
blonde, cheap, yes, reject.
grey, expensive, yes, accept.
blonde, expensive, yes, accept.
black, expensive, yes, accept.
brown, expensive, no, reject.
blonde, average, yes, reject.
blonde, expensive, no, accept.

Same display but manually:
black, cheap, yes, reject.
black, cheap, no, reject.
grey, average, no, reject.
blonde, cheap, yes, reject.
grey, expensive, yes, accept.
blonde, expensive, yes, accept.
black, expensive, yes, accept.
brown, expensive, no, reject.
blonde, average, yes, reject.
blonde, expensive, no, accept.
```

Figure 15: The output from "readil.c."

```
// Tutorial program : accuracy estimation

#include <basics.h>
#include <ID3Inducer.h>
#include <NaiveBayesInd.h>
#include <CtrInstList.h>
#include <AccEstDispatch.h>

main()
{
   ID3Inducer    inducerDT("Xema-DT"); // Decision tree inducer
   NaiveBayesInd inducerNB("Xema-NB"); // Naive-Bayes inducer

   CtrInstanceList bag("breast");
   AccEstDispatch accEst;
   accEst.set_user_options("");

   // Keep the inducer quiet since we'll be running it many times
   inducerDT.set_log_level(max(globalLogLevel - 3, 0));
   accEst.estimate_accuracy(inducerDT, bag);
   Mcout << "Estimated accuracy for decision tree\n" << accEst << endl;

   inducerNB.set_log_level(max(globalLogLevel - 3, 0));
   accEst.estimate_accuracy(inducerNB, bag);
   Mcout << "Estimated accuracy for naive-bayes\n" << accEst << endl;

   return 0;
}
```

Figure 16: The program "ae.c" for estimating the accuracy of ID3 and Naive-Bayes

to display themselves in a reasonable manner. The rest of the program shows how to do this by getting each instance and printing it. We use the fact that an instance knows how to print itself. Pixes are pseudo indexes. These are used to iterate through container classes such as a `InstanceList`. You can call the member functions `first()`, `next()`, `last()`, `prev()` when using pixes. Given a pix, `get_instance()` will get you the instance.

7 Accuracy Estimation

Given a learning algorithm (an inducer), $\mathcal{MLC}$++ provides *wrappers,* which "wrap around" a given inducer to provide additional functionality. One family of wrappers are accuracy estimation wrappers that attempt to estimate the future prediction accuracy of a categorizer built by an inducer using a given dataset.

Figure 16 shows a program that estimates the accuracy of two induction algorithms, ID3 and Naive-Bayes, on a dataset. Using these method, one can attempt to automatically pick the best inducer for the task at hand. As Schaffer (1993) writes, accuracy estimation methods such as cross-validation are likely to correctly predict the better categorizer if the differences are significant.

```
MLC++ Debug level is 1, log level is 1
OPTION PROMPTLEVEL = required-only
OPTION REMOVE_UNKNOWN_INST = false
OPTION CORRUPT_UNKNOWN_RATE = 0
Reading breast.data..... done.
OPTION ACC_ESTIMATOR = cv
OPTION ACC_EST_SEED = 7258789
OPTION ACC_TRIM = 0
OPTION CV_FOLDS = 10
OPTION CV_TIMES = 1
OPTION CV_FRACT = 1
Inducer: Xema-DT
Number of folds: 10, Number of times: 1
Untrimmed accuracy 95.28% +- 0.95% (89.13% - 97.87%)
Estimated accuracy for decision tree
Method: cv
Trim: 0
Seed: 7258789
Folds: 10,  Times: 1
Accuracy: 95.28% +- 0.95% (89.13% - 97.87%)

Inducer: Xema-NB
Number of folds: 10, Number of times: 1
Untrimmed accuracy 95.05% +- 0.73% (91.30% - 97.87%)
Estimated accuracy for naive-bayes
Method: cv
Trim: 0
Seed: 7258789
Folds: 10,  Times: 1
Accuracy: 95.05% +- 0.73% (91.30% - 97.87%)
```

Figure 17: The output from "ae.c."

7.1 Speeding Up the Execution

By default $\mathcal{MLC}$++ runs in debug mode which attempts to catch programmer mistakes. After a program is debugged on a small dataset with small parameters, it is possible to test it on a larger dataset and larger parameters by turning the debug mode off.

The environment variable `DEBUGLEVEL` determines whether debugging takes place or not. If this environment variable is not defined, it defaults to `INT_MAX`. Defining it to 0 eliminates all debugging, and using small numbers such as 1 turns some debugging on.

To run a program `a.out` faster, you can type the following:

```
setenv DEBUGLEVEL 0
a.out
unsetenv DEBUGLEVEL
```

Avoiding the debug statements can decrease running time to about one third. Further improvements can be achieved by using the "fast" $\mathcal{MLC}$++ library which is compiled without all the debug statements themselves, and with some inlined functions. The fast library usually decreases running time to about one fifth of the original running time.

> It is strongly suggested that `DEBUGLEVEL` be set to 0 only after the program has been thoroughly debugged.

8 Summary

The tutorial is a basic introduction to $\mathcal{MLC}$++. The library has other options and classes not mentioned here. The classes shown usually have many other options not mentioned, different constructors, and other functions.

Writing machine learning algorithms is not an easy task, but testing them on a wide spectrum of datasets, identifying their strengths and weaknesses, and plotting learning curves makes this task even harder. $\mathcal{MLC}$++ provides the evaluation tools that are needed for many learning algorithms, and those tools that can help make writing a learning algorithm easier. Learning any new tool imposes extra overhead, but those who write everything from scratch are doomed to progress very slowly.

$\mathcal{MLC}$++ is an ongoing project that is constantly being updated and improved. Suggestions, comments, and criticisms may be sent to `mlc@cs`.

A Troubleshooting

This appendix describes common problems that may occur using $\mathcal{MLC}$++. In case you cannot solve a problem and need help, you can send e-mail to `mlcpp-help@CS.Stanford.EDU`. If you think you have found a bug, send e-mail to `mlcpp-bugs@CS.Stanford.EDU`.

A.1 Setup Problems

- The file compiles, but running the setup test produces the following message:

  ```
  Error - MLCIStream::open_read: Could not open file
          monk1-full.names or ...
  ```

  The setup test opens the file monk1-full.names. It searches for it in the current directory, and then in $MLCPATH. The script `$MLCDIR/setup` should have defined it appropriately. Try running the script, *i.e.*,

  ```
  source $MLCDIR/setup
  ```

- When attempting to compile, the compiler generates many errors beginning with

  ```
  setupTest.c:9: basics.h: No such file or directory
  ```

  The problem is that the compiler is unable to access $\mathcal{MLC}$++ include files. The command `MLCC` is aliased in `$MLCDIR/aliases`. Try running the script, *i.e.*,

  ```
  source $MLCDIR/aliases
  ```

A.2 Compiling

The compiler does not give any warnings or errors, but the `a.out` file is not generated.

Due to a bug in the CC compiler, you must create a directory called "ptrepository" in the directory you are working. $\mathcal{MLC}$++ uses double template repositories, and if your programs will need to instantiate templates, the directory must exist. To create the directory, type the following

```
mkdir ptrepository
```

A.3 Fatal Errors and Core Dumps

By default, fatal errors caught by $\mathcal{MLC}$++ abort the executing program. The default behavior is to print the message and exit with a bad status. In some cases, the error message is not obvious, or is triggered by a low level function. In such cases, it may aid the programmer to look at the call stack.

The default behavior can be changed by setting the environment variable `DUMPCORE` to `yes`. Fatal errors will now generate a core dump which can be analyzed using debuggers. Note that in order to dump the core, the core file must be writable (*i.e.*, core will not be dumped if you do not have write permission to the directory and to the core file). Furthermore, you must not have a limit on the core dump size. From `csh`, typing `limit` should show that `coredumpsize` is unlimited. If it is limited, type `unlimit coredumpsize`.

$\mathcal{MLC}$++ will recognize when it is run from ObjectCenter, and will always call `abort()` instead of `exit()` on a fatal error. This behavior will allow going up the call stack and displaying variables and function arguments.

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.

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. (1994), Bottom-up induction of oblivious, read-once decision graphs : strengths and limitations, *in* "Twelfth National Conference on Artificial Intelligence", pp. 613–618.

Kohavi, R. (1995), 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.

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

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. (1995), UCI repository of machine learning databases, http://www.ics.uci.edu/~mlearn/MLRepository.html.

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.

Schaffer, C. (1993), "Selecting a classification method by cross-validation", *Machine Learning* **13**(1), 135–143.

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

Ullman, J. D. (1988), *Principles of database and knowledge-base systems*, Vol. 1, Computer Science Press.

Wnek, J., Sarma, J., Wahab, A. A. & Michalski, R. S. (1990), Comparing learning paradigms via diagrammatic visualization, *in* "Methodologies for Intelligent Systems, 5. Proceedings of the Fifth International Symposium", pp. 428–437. Also technical report MLI90-2, University of Illinois at Urbaba-Champaign.

Index