Overview of $\mathcal{MLC}$++ Library Classes

Richard Long
long@cs.Stanford.EDU

April 19, 1994

Contents

1 Introduction

This document gives a brief overview of all of the classes in the $\mathcal{MLC}$++ library. It is not intended to give an exhaustive explanation of each function or details of implementation, but rather a general idea of how to use each class in the common case. For further details on a specific class see the reference manual or the appropriate ".c" file.

1.1 Intended Audience

This document assumes that the reader has a basic familiarity with machine learning and a good understanding of C++.

1.2 Conventions and General Comments

Throughout this document, C++ class names and function calls will appear in `typewriter style`. All of the classes are concrete (instantiable) unless otherwise stated. For brevity, it is not in general stated whether the functions or arguments are `const`. These syntactic details are available in the reference manual as well as in the header files.

1.3 Related Documentation

The following documents are currently available for readers who want a better understanding of $\mathcal{MLC}$++:

cc-coding 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 issues of design.

mlc-coding $\mathcal{MLC}$++ coding standards. These are conventions used in $\mathcal{MLC}$++.

environ The $\mathcal{MLC}$++ environment. This includes information about 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.

GLD-design Description of the design considerations and extensions.

dotdoc Documentation describing the `dot` program which is used by $\mathcal{MLC}$++ to display trees and graphs.

tutorial Documentation and exercises which introduce $\mathcal{MLC}$++ to a new user.

2 Overview of Core Classes and Functions

$\mathcal{MLC}$++ has a number of classes and functions that are used throughout the library that are not machine learning specific.

2.1 Error Handling

$\mathcal{MLC}$++ performs extensive error checking. `fatal_error` is used to report errors to the user. `fatal_error` is a stream manipulator and should be used with the stream `err`. The syntax is:

```
err << "class::function: bad value: " << val << fatal_error;
```

Such a statement will cause the program to abort and will flush the given message (with automatic line wrapping) to the standard error output.

$\mathcal{MLC}$++ has also provides a `new_handler` that causes a `fatal_error` if a call to `new` fails. Thus, whenever a file is linked properly with the $\mathcal{MLC}$++ library, there is no need to check for a failure after calling `new`.

2.1.1 DBG() and ASSERT()

`DBG()` is a macro that allows the program to avoid executing certain time consuming code, while allowing the developer to retain error checking for debugging. `DBG()` executes the enclosed code if the environment variable `DBGLEVEL` is greater than zero and the compile time define `FAST` is not defined. Code that is executing when `DBGLEVEL` is zero or when `FAST` is compile time defined is said to be executing in "fast" mode. `DBG()` is often used with error checking code, so that extensive error checking is done while testing and debugging code, but those checks can be easily avoided when the debugged code is run on larger problems. Two variations on `DBG()` are `DBGDECLARE()` and `DBGSLOW()`. `DBGDECLARE()` should be used to declare variables that will not be needed when the code is executing in "fast" mode. `DBGSLOW()` should only be used for *very* slow code; the only difference from `DBG()` is that code is not executed if `DBGLEVEL` is less than 2.

The `ASSERT()` macro is used to perform internal consistency checks. The user should not see these errors; a triggered `ASSERT()` indicates a bug in the internal $\mathcal{MLC}$++ code (as opposed to a bad argument or other "user" problem). (`ASSERT()` uses `DBG()` and is thus removed from the code under the same conditions as `DBG()` code.)

2.2 basics.h

"basics.h" is a file which declares many of the basics types and functions in $\mathcal{MLC}$++. It should be included first in every ".c" file.

2.2.1 Types

`Bool` is a boolean type with values `TRUE` and `FALSE`. `Real` is the type used in $\mathcal{MLC}$++ to store floating point numbers. Several constants about maximum and minimum values for numerical types are also declared in "basics.h."

2.2.2 Other useful functions

`get_env(MString envVarName)` returns the string value of the environment variable with the given name or causes a `fatal_error` if the named environment variable does not exist.

`get_env_default(MString envVarName, MString defaultValue)` behaves like `get_env()` except that it returns `defaultValue` instead of causing a `fatal_error` if the named environment variable is not defined.

`min(a, b)` and `max(a, b)` are template functions that return the minimum and maximum respectively of the two arguments. `min()` requires that `operator<=` be defined for the arguments, and `max()` requires that `operator>=` be defined for the arguments.

2.3 I/O and MLCStreams

$\mathcal{MLC}$++ provides stream classes to aid in input and output. `MLCIstream` is an input file stream. `MLCOstream` provides output support for file streams, and graphical streams.

2.3.1 File Streams

To handle input and output, $\mathcal{MLC}$++ provides "safe" file streams in that they check for errors after each operation is called. Function calls abort if the status of the stream is bad. To perform "safe" output use the macro `MLCOUT(MLCOstream&, cmd)`, where `cmd` is a sequence of arguments that would be sent to a stream using `operator<<`, with the initial `<<` omitted. For example, to display the output of an age variable, one might write:

```
MLCOUT(Mcout, "Age: " << age << endl);
```

`MLCIN(MLCIstream&, cmd)` is the analogous input macro. `Mcout`, `Mcin`, and `Mcerr` are `MLCStreams` declared in "basics.h" that correspond to the C++ streams `cout`, `cin`, and `cerr` respectively.

The `MLCIstream` and `MLCOstream` constructor takes the (non empty) name of the file that it should open. The file is closed when the the class instance goes out of scope. Calling `description()` returns the name of the file with which the stream is associated (or an appropriate description if it is `Mcout`, `Mcin`, or `Mcerr`).

2.3.2 Output Types

By default, streams are associated with files, but MLC++ also provides support for X streams. Certain MLC++ function can display output to windows in X. These functions require that the `MLCOstream` have `OutputType XStream`. This is simply an enumerated type that is an optional second argument to the `MLCOstream` constructor. The default `OutputType` is `FileStream`.

2.4 MString

`MString` is the MLC++ implementation of strings that do their own memory allocation, know their length, and also provide some other useful features. `MString` provides a conversion operator that allows it to be used whenever a `const char*` is needed. `MStrings` can be constructed using `char*, char, Real` (with precision), or `long, int,` or `short` with a width. `length()` returns the length of the `MString`. `MStrings` can be concatenated using `+, +=, cat(),` and `prepend()`. Input and output of `MStrings` is supported through `operator>>` and `operator<<`. `MStrings` can be compared using operators `==, !=, <, <=, >,` and `>=` (no need for `strcmp()`). Note that if you compare an `MString` and a `char*`, the `char*` will be converted to an `MString` automatically, and the operation will return the expected result. If an `MString` represents a `Real, long, short,` or `char`, it can be converted using `real_value(), long_value(), short_value()` and `char_value()` respectively.

2.5 Arrays

The MLC++ template class `Array` is a one-dimensional array that knows its size and provides bounds checking (except in "fast" mode (See Section 2.1.1)). An `Array` can be instantiated by providing either the size of the array (in which case the index will be zero based), or the lower bound and size (*not* upper bound) of the array. In the latter case, an optional argument to the constructor is the initial value for all elements. `low()` and `high()` return the lower and upper bounds of the array respectively. `size()` returns the size of the array. Elements of the array are accessed using `operator[]`. Another option is to use `index(int)`, which indexes elements as if the lower bound were zero. `max()` and `min()` return the maximum and minimum elements of the array if `operator>` and `operator<` respectively are defined for the class with which `Array` is instantiated.

`PtrArray` is a subclass of array that should only be instantiated with pointer elements. It differs from `Array` in that it initializes all of the elements to `NULL` and its destructor deletes all objects to which its elements point.

The template class `Array2` is a two-dimensional array. The constructors take either the number of rows and number of columns, or the start row, start column, number of rows, and number of columns, and optionally an initial value for all elements.

The Array2 size is provided by num_rows(), num_cols(), start_row(), start_col(), high_row(), high_col(), and size(). size() returns the number of rows times the number of columns. Elements of the array are accessed using operator() with the row and column as arguments. index() accesses elements as if it were a one-dimensional array with upper bound size()-1 and lower bound zero. PtrArray2 has the same relationship to Array2 that PtrArray has to Array.

2.6 Doubly Linked List and DLLPix

The class DblLinkList is a doubly linked list template class. The template argument to DblLinkList is the type to be stored in the list. (This argument will be called T for the remainder of this section.) The given type must provide a copy constructor. The DblLinkList constructor creates an empty list. DblLinkList provides functions one would expect for a list:

- front() returns the first element in the list.
- rear() returns the last element in the list.
- remove_front() deletes the first element in the list and returns the deleted element.
- remove_rear() deletes the last element in the list and returns it.
- del_front() deletes the first element of the list, but does not return it.
- del_rear() deletes the last element in the list without returning it.
- length() returns the number of elements in the list.
- display(MLCOstream) and operator<< display the elements of the list if T has operator<< defined.

A pix (Pseudo IndeX) is used to iterate through elements of a container. DLLPix is a pix template class for DblLinkList. A DLLPix indexes an element of a given instance of a DblLinkList. DLLPix also has a special "clear" value indicating that it does not index any element of the list. The specific element that the DLLPix indexes, however, is not fixed—it will vary as the pix iterates through the list. The template argument to DLLPix should be the same T used for the associated DblLinkList. The DLLPix constructor takes a DblLinkList as an argument. The second argument to the DLLPix constructor determines the initial value of the pix: -1 sets the pix to the last element in the list, 0 sets the pix to be clear, and 1 sets the pix to the first element in the list. The second argument defaults to 0. The following are functions of the DLLPix:

- first() and last() set the pix to the first and last items respectively in the associated DblLinkList.

- next() and prev() move to other items in the list. Calling next() when the pix indexes the last element in the list sets the pix to "clear." Similarly, when prev() is called and the pix indexes the first element of the list, the pix is set to "clear."

- operator* returns a const reference to an item in the list.

- is_clear() indicates whether the pix is clear. As an alternative to is_clear(), operator Bool and operator! are provided.

- operator== and operator!= allow for comparison of two pixes. Note that it is an error to compare invalid or clear pixes.

DblLinkList provides several functions that use DLLPix for inserting elements:

- DLLPix<T> ins_before(DLLPix<T>, T) inserts the given element before the element indicated by the pix.

- DLLPix<T> ins_after(DLLPix<T>, T) inserts the given element after the element indicated by the pix.

- DLLPix<T> prepend(T) inserts the given element at the front of the list.

- DLLPix<T> append(T) inserts the given element at the end of the list.

All the insertion functions return a pix indexing the newly inserted element.

Other DblLinkList functions use DLLPix to access and delete elements:

- operator()(DLLPix<T>) returns the (not necessarily const) element in the list indicated by the given pix.

- owns(DLLPix<T>, Bool fatal_on_false) returns TRUE if and only if the list contains the element indexed by the pix. It aborts on FALSE if the fatal_on_false argument is TRUE.

- del(DLLPix<T>, int dir) deletes the element indexed by the given pix. If dir is 1 the pix indexes next element and if dir is -1 pix indexes the previous element.

- del_after(DLLPix<T>) deletes the element in the list that follows the element indexed by the given pix.

Before DLLPix is instantiated, the macro SET_DLLPIX_CLEAR(dataType, pixClearVal) should be used to set the "clear" value for the DLLPix. pixClearVal is the value (typically NULL) given to a clear pix. See Figure 1 for an example of a DLLPix in use.

```
SET_DLLPIX_CLEAR(MString,NULL);

int foo(const DblLinkList<MString>& names, const MString& wanted)
{
   int numMatches = 0;
   for (DLLPix<MString> pix(names, 1); pix; pix.next())
      if (names(pix) == wanted)
         numMatches++;
   return numMatches;
}
```

Figure 1: The function foo shows a typical use of DLLPix.

2.7 Random Numbers

`MRandom` is an $\mathcal{MLC}$++ class for generating random numbers. The constructor can optionally take an `unsigned` seed. The seed can be reset using `init(unsigned)`. Random numbers are generated using the functions:

- `long_int(long)`
- `long_int(long low, long high)`
- `integer(int)`
- `integer(int low, int high)`
- `real(Real low, Real high)` returns a `Real` in range [`low`, `high`).
- `boolean()` returns `TRUE` or `FALSE`.

2.8 MOptions

`MOptions` is a class that provides a standard set of calls for logging information. $\mathcal{MLC}$++ classes which support logging having a public `MOptions` (or derived class) member called `options`. The log level can be set and retrieved using `set_log_level()` and `get_log_level()` respectively. The higher the log level, the more logging information is given. Level 0 provides no logging information. The `MLCOstream` that receives the logging messages can be set and retrieved using `set_stream()` and `get_stream()`.

2.9 Ownership

Ownership is a concept that is used in $\mathcal{MLC}$++ for memory management. The owner of an object is responsible for deallocating it. It is possible to allocate an object and transfer ownership to a different entity which will be in charge of deallocating it. Ownership is recursive to all levels unless specified differently, *i.e.*, ownership of a tree implies ownership of strings pointed to from the nodes.

When a function gets ownership of an object, the function should declare the argument as `ptr*&`. This allows the function to copy the pointer and also set the caller's pointer to NULL, so that the caller will not use it again. This is intended to avoid mistakes more than a safe-proof method, as the caller can copy the pointer in advance.

3 Basic Bags and Lists

Bags and lists are containers for instances that serve as databases to train and test inducers. The main distinction between bags and lists is that lists are ordered, whereas bags are not.

3.1 LabelledInstanceBag

The argument to the `LabelledInstanceBag` (labelled instance bag) constructor is a `LabelledInstanceInfo` (see Section 4.2) which specifies the type of instance that the bag will contain. `labelled_instance_info()` returns the `LabelledInstanceInfo` for the bag. Instances are added to the bag using `add_labelled_instance()`. Instances can be removed using `remove_labelled_instance(Pix)` and `del_labelled_instance(Pix)`. To iterate through instances in the bag, use `first()`, `next(Pix)`, and `get_labelled_instance(Pix)`. `find(Instance)` returns a `Pix` for a labelled instance in the bag that matches the given instance. `num_labelled_instances()` and `no_labelled_instances()` can be used to find the number of labelled instances in the bag. `num_categories()` returns the number of possible categories that an instance in the bag can have. The bag can be displayed to an `MLCOstream` using `display(MLCOstream)` or `operator<<`.

`LabelledInstanceBag` provides functions to divide its contents based on some criteria. `split_by_label()` returns an array of bag pointers. For each bag, all instances in that bag have the same label. `split(Categorizer&)` also returns an array of bag pointers with the split based on the `categorize()` function of the given `Categorizer` (see Section 5).

`independent_sample(int size)` returns a bag containing an independent sample of instances of the given size. `shuffle()` reorders the bag.

3.2 LabelledInstanceList

`LabelledInstanceList` (labelled instance list) is derived from `LabelledInstanceBag` (see Figure 2). Instance lists can be read from files using the constructor which takes a file stem as an argument. Reading a list requires a "names" file and a "data" file. (See "InstList.c" or the Reference Manual for file format.) To read a new group of labelled instances into a previously instantiated `LabelledInstanceList` call `read_data()` providing the "data" file name as an argument. Calling `read_data()` will delete any instances previously in the bag.

Because lists have order, they have additional functions to access instances that use this order. `del_front()` and `remove_front()` remove and return the first instance in the list. `split_prefix(int)` removes the given number of instances from the list and returns them in a new list. `unite(LabelledInstanceList*)` appends the instances from the given list to the called list.

3.3 LabelledCtrInstanceBag, LabelledCtrInstanceList, and BagCounters

In some cases it is convenient and efficient to have counter information for the instances in a list or a bag. This functionality is provided by the class `BagCounters`. `BagCounters` works only for nominal attributes and will abort otherwise. It has the following functions:

- `val_count(NominalVal labelVal, int attrNum, NominalVal attrVal)` returns the number of instances with the indicated label that have the given attribute value for the indicated attribute.
- `attr_count(int attrNum, NominalVal attrVal)` returns the number of instances that have the given attribute value for the indicated attribute.
- `label_count(NominalVal labelVal)` returns the number of instances that have the given label.
- `attr_num_vals(int attrNum)` returns the number of attribute values that exist in the bag for the indicated attribute.
- `label_num_vals()` returns the number of different existing labels in the bag.
- `display(MLCOstream)` and `operator<<` display the `BagCounters`.

The classes `LabelledCtrInstanceBag` (labelled counter instance bag) and `LabelledCtrInstList` (labelled counter instance list) each have `BagCounters` which can be accessed using `counters()`. `ctr_split(Categorizer)` is the same as `split(Categorizer)`, except it returns an array of counter bags.

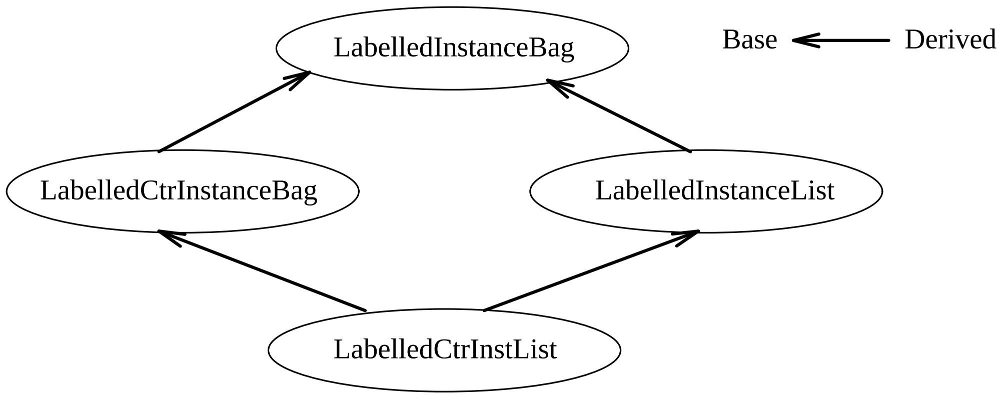

Figure 2: The Bag-List inheritance hierarchy.

3.4 Polymorphism with Bags and Lists

Because `LabelledInstanceList`, `LabelledCtrInstanceBag`, and `LabelledCtrInstList` have the common base class `LabelledInstanceBag` (see Figure 2), an object of any of these classes can be used in cases where a `LabelledInstanceBag` is required. Because type casting is not safe in general, and it is complicated by multiple inheritance, $\mathcal{MLC}$++ provides the functions `cast_to_labelled_instance_list()`, `cast_to_labelled_ctr_instance_bag()`, and `cast_to_labelled_ctr_inst_list()`. These functions return a reference to the appropriate type if the cast is legal and cause a `fatal_error` if the cast is illegal.

4 Instances and Attributes

Machine learning instances are represented in $\mathcal{MLC}$++ by the class `Instance`. Conceptually, an instance is a collection of attributes values and the associated attribute information (See Figure 3). Instances which have an associated label are `LabelledInstances`. $\mathcal{MLC}$++ supports a variety of types of attributes which will be described in this section.

4.1 Instance and LabelledInstance

`Instance` and `LabelledInstance` are generally not created directly by users; usually, they are retrieved from a bag (see Section 3) or returned by a function call to a `Categorizer` (see Section 5) or `Inducer` (See Section 6). `attr_info(int)` returns the `AttrInfo` (See Section 4.3.2) indexed (starting from zero) by the given integer. `attr_name(int)` returns a description of the indicated attribute. `instance_info()` returns the `InstanceInfo` (See Section 4.2) for the `Instance`. `operator[]` returns the value of the indicated attribute. `Instance` can be displayed using `display(MLCOstream)` or `operator<<`.

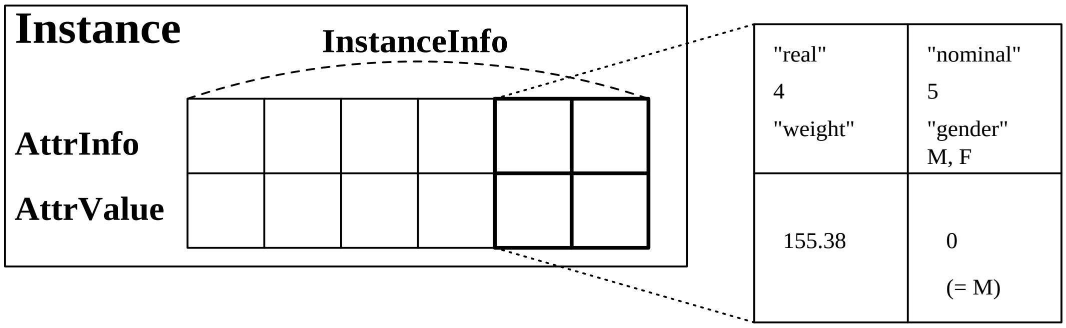

Figure 3: Conceptual diagram of an instance. The fourth and fifth Attributes (indexed from zero) are shown in greater detail. Attribute four has description "weight" and is of type real with value 155.38. Attribute five has description "gender" and is of type nominal with value "M".

A `LabelledInstance` is an instance which also has an attribute that describes its label. The functions `attr_info(int)`, `attr_name(int)`, and `operator[]` are the same as for `Instance`. `label_info()` returns the `AttrInfo` for the label, and `label()` returns the value of the label. `labelled_instance_info()` returns the `LabelledInstance-Info` for the `LabelledInstance`. `display(MLCOstream)` and `operator<<` both print a `LabelledInstance` to the indicated stream. `operator Instance` automatically converts a `LabelledInstance` to an `Instance`, so a `LabelledInstance` can be used anywhere that a `Instance` is required.

4.2 InstanceInfo and LabelledInstanceInfo

`InstanceInfo` is a collection of `AttrInfo` for all of the attributes in an instance (See Figure 3). `InstanceInfo` allows the attribute values of an instance to be interpreted correctly.

The `InstanceInfo` constructor creates an object with no `AttrInfo`. `add_attr(AttrInfo*&)` adds an `AttrInfo` to the end of the `InstanceInfo`. `num_attr()` returns the number of `AttrInfo`s in the `InstanceInfo`. The pix operations `first_attr()`, `last_attr()`, `next_attr()`, and `prev_attr()` index the `AttrInfo`s. `attr_info(Pix)` returns the `AttrInfo` indexed by the given pix. `attr_info(int)` returns the `AttrInfo` whose `num()` (See Section 4.3.2) matches the given integer. `InstanceInfo`s can be compared using `operator==` and `operator!=`. `display(MLCOstream)` and `operator<<` print the `InstanceInfo` to the given stream.

`LabelledInstanceInfo` has an additional `AttrInfo` that is reserved for the label of the instance. It is set and retrieved using `set_label_info(AttrInfo*&)` and `label_info()` respectively. All other functions are the same as `InstanceInfo`.

4.3 AttrInfo and AttrValue

Attributes are the basic building blocks of instances. The values for attributes are stored using the class `AttrValue`. The `AttrInfo` class hierarchy is used to describe the type of information that is stored in an `AttrValue`.

4.3.1 AttrValue

`AttrValue` is a union of real and integer values using the `realVal` and `intVal` fields respectively. `AttrValue` is not self-describing. To interpret the value, one must use the associated `AttrInfo`.

4.3.2 AttrInfo

`AttrInfo` is an abstract base class that defines the basic interface for its subclasses. Each `AttrInfo` has a name set by the constructor that can be retrieved with `name()`. Since `AttrInfos` are usually members of an `InstanceInfo` (See Section 4.2), they are also aware of their position, which is accessed through `num()`. `check_in_range(AttrValue&)` causes a fatal error if the given value is not legal for the `AttrInfo`. Input and output of `AttrValues` is done using `AttrInfo`. `attrValue_to_string(AttrValue&)` returns an `MString` representation of the given `AttrValue`. `read_attr_value(MLCIstream&, int& line)` reads an `AttrValue` from the given `MLCIstream`. `AttrInfos` can be displayed using the functions `display(MLCOstream)` and `operator<<()`. `display_values(MLCOstream)` prints the `AttrInfo` in "names" file format (See Section 3.2). `AttrInfo` also provides `operator==()` and `operator!=()`. `is_unknown(AttrValue)` returns `TRUE` if and only if the given `AttrValue` is unknown.

4.3.3 NominalAttrInfo

`NominalAttrInfo` is a subclass of `AttrInfo` that allows the user to enumerate the different values that the attribute may obtain. The constructor takes as arguments a description of the attribute, its number, an `MStringList` of the values it can attain, and an `AttrType`, which defaults to `nominal`. (`AttrType` is an enumerated type defined in "Attribute.h"; the only time to use a type other than the default is when subclassing from `NominalAttrInfo`.) Rather than typecasting to convert an `AttrInfo` into a `NominalAttrInfo`, use the member function `cast_to_nominal()`, which only performs the cast if the object truly is a `NominalAttrInfo`, and causes a `fatal_error` otherwise. To check if an `AttrInfo` is nominal, call `can_cast_to_nominal()`.

A number of functions give information about the possible values that a nominal attribute can attain:

- `num_values()` returns the number of possible values.

- get_value(int i) returns the string representation of the ith nominal value; the argument should be in the range [0, num_values()-1].

- nominal_to_int(MString) returns the index into the NominalAttrInfo that corresponds to the given string representation of the attribute value.

4.3.4 RealAttrInfo

RealAttrInfo is derived from AttrInfo and provides support for attribute values which have real values. The RealAttrInfo constructor takes a name, its number, and an AttrType as arguments. (AttrType defaults to real, and should only be changed for subclasses.) To check if an AttrInfo is real, use can_cast_to_real(). A number of functions can be used to compare two AttrValues with the same RealAttrInfo:

- less_than(AttrValue, AttrValue)
- less_than_equal(AttrValue, AttrValue)
- greater_than(AttrValue, AttrValue)
- greater_than_equal(AttrValue, AttrValue)
- equal(AttrValue, AttrValue)
- not_equal(AttrValue, AttrValue)

Note that these are not the standard C++ comparison operators; they are called using a RealAttrInfo with AttrValue arguments. distance(AttrValue, AttrValue) returns the absolute value of the difference of the two real valued AttrValues. The distance from an unknown AttrValue is undefined, and causes a fatal_error.

5 Categorizers

A categorizer is used to predict the category for a instance. (These are often called classifiers in the literature, but $\mathcal{MLC}$++ uses categorizer and category as opposed to classifier and class to avoid confusion with the C++ class.) Categorizer is an abstract base class that defines a standard interface for its subclasses. The constructor takes the number of possible categories and a description as arguments. num_categories() and description() returns these values. categorize(Instance) returns the categorizer's predicted category for the given instance. display_struct(MLCOstream, DisplayPref) prints the structure that the categorizer uses to make its prediction.

5.1 AugCategory

AugCategory is a category with the additional information of a string description of its value. The constructor takes the category value and a description of that value. num() returns the integer value of the category. description() returns the MString. operator Category provides automatic conversion from AugCategory to Category. display(MLCOstream) and operator<< print the AugCategory to the indicated stream. AugCategoryAlloc is a subclass of AugCategory that owns its description (See Section 2.9).

5.2 ConstCategorizer

ConstCategorizer (constant categorizer) always returns the same category, regardless of the given instance. The constructor requires a description of the categorizer and an AugCategoryAlloc that will be the value returned from categorize().

5.3 AttrCategorizer

AttrCategorizer (attribute categorizer) categorizes an instance based on the value of a single attribute. The constructor requires an InstanceInfo for the type of instances that it will categorize, the index into that InstanceInfo (See Section 4.3.2), and a description.

5.4 TableCategorizer

The TableCategorizer (table categorizer) constructor requires a labelled instance bag and a description as arguments. To categorize, the table categorizer looks in its labelled instance bag for a matching instance. It returns the label of the first matching instance, or UNKNOWN_CATEGORY_VAL if no matching instance is found.

5.5 BadCategorizer

BadCategorizer is used in cases where a categorizer should never be called. All of the Categorizer functions except for the constructor, destructor, and description() cause fatal_errors. BadCategorier::is_bad_categorizer(Categorizer) returns TRUE if and only if the given categorizer is a bad categorizer. A static BadCategorizer called badCategoerizer is instantiated in "BadCategorizer.c" and externed in "BadCategorizer.h." It is an error to try to instantiate another BadCategorizer.

5.6 CatGraph, RootedCatGraph, and DecisionTree

CatGraph is a directed, acyclic graph (DAG) whose nodes are categorizers and whose edges are AugCategorys. The edges are numbered sequentially starting at UNKNOWN_

CATEGORY_VAL or FIRST_CATEGORY_VAL. A CatGraph is permitted to have disconnected components.

The CatGraph constructor creates a graph with no nodes or edges. create_node(int level) and create_node(Categorizer*&, int level) return a NodePtr to a newly created node. The level argument is optional, and should be specified only for levelled graphs. level(NodePtr) returns the level of the given node. If the categorizer is specified, then the node contains that categorizer. If no categorizer is specified, the node contains a pointer to badCategorizer (See Section 5.5). set_categorizer(NodePtr, Categorizer*&) sets the categorizer for a node which has not yet been set. It is an error to set a node more than once or to set a node to a BadCategorizer. get_categorizer(NodePtr) returns the categorizer associated with the given node. connect(NodePtr from, NodePtr to, AugCategory edgeLabel) creates an edge from node from to node to with the given edge label. get_child(NodePtr parent, AugCategory edgeLabel) returns the child of the given node along the edge with the given label. num_children(NodePtr) returns the number of children for the given node. check_reachable(NodePtr source) returns TRUE if and only if all nodes in the graph can be reached from the given node. check_tree(NodePtr root) returns TRUE if and only if the graph forms a tree and the given node is the root. check_node_in_graph(NodePtr) returns TRUE if and only if the given node is in the graph. num_nodes() and num_leaves() return the number of nodes and leaves respectively in the graph. display(MLCOstream, DisplayPref) and operator<< display the graph to the given stream.

RootedCatGraph is a CatGraph with a specified root node from which all nodes are reachable. The root is set and returned using set_root(NodePtr) and get_root() respectively. DecisionTree is a RootedCatGraph in which the root node has no parents and each other node has exactly one parent. The OK() function checks that a CatGraph, RootedCatGraph, or DecisionTree has a state consistent with its specification.

5.7 RDGCategorizer

RDGCategorizer (rooted decision graph categorizer) uses a RootedCatGraph to categorize instances. The constructor requires a RootedCatGraph*, a description, and the number of possible categories. rooted_cat_graph() returns the RootedCatGraph used for categorizing. num_nodes() and num_leaves() return the number of nodes and leaves respectively in the categorizer's RootedCatGraph.

5.8 DTCategorizer

DTCategorizer (decision tree categorizer) is derived from RDGCategorizer with the difference that it uses a DecisionTree to categorize.

6 Inducers

The Inducer class *induces* a concept from a labelled training set (supervised learning). In $\mathcal{MLC}$++ terms, an Inducer creates a Categorizer using a LabelledInstanceBag. Inducer is an abstract base class that defines the standard interface for all inducers:

- description() returns a description of the inducer.
- read_data(MString name, MString namesExt, MString dataExt) reads the training set for the inducer from the files with the given name and extensions. The files should have the LabelledInstanceList file format.
- assign_data(newTS) assigns the training set for the Inducer to be newTS, a pointer to an already existing instance from the LabelledInstanceBag hierarchy. assign_data(newTS) also returns a pointer to the previous training set, which may be NULL.
- has_data() indicates whether the inducer has a training set.
- labelled_instance_bag() returns a reference to the training set.
- train() uses the training set to build a Categorizer.
- was_trained() indicates whether train() has been called for the inducer.
- get_categorizer() returns the categorizer that has been constructed for the inducer.
- predict(Instance) returns the predicted category for the instance.
- display_struct(MLCOstream, DisplayPref) prints the induced structure to the given stream.

CtrInducer is a subclass of Inducer (but still an abstract base class) that provides counter operations on the training set. It requires that the argument to assign_data() be a LabelledCtrInstanceBag or LabelledCtrInstList. TS_with_counters() returns a LabelledCtrInstanceBag for the training set. counters() returns the BagCounters for the training set.

6.1 ConstInducer

ConstInducer is a subclass of Inducer that induces a constant categorizer. It uses the majority category of the training set to predict all instances.

6.2 TDDTInducer and ID3Inducer

`TDDTInducer` (top-down decision-tree inducer) induces a decision tree top-down by building smaller training sets and inducing decision trees for them recursively. `TDDTInducer` is a subclass of `CtrInducer` (but still an abstract base class). `num_nodes()` and `num_leaves()` return the number of nodes and leaves respectively in the induced decision tree.

`ID3Inducer` is a subclass of `TDDTInducer` that uses the ID3 algorithm to induce the decision tree [Qui86].

6.3 OODGInducer and HOODGInducer

`OODGInducer` (Oblivious, read-Once Decision Graph inducer) is an abstract base class and a subclass of `Inducer`. `OODGInducer` uses a bottom-up method for building levelled rooted decision graphs [Koh94]. If the environment variable `OODGREMOVEDUP` (which defaults to "yes") is set to "yes," conflicting and duplicate instances are removed. `num_nodes()` and `num_leaves()` return the number of nodes and leaves respectively in the induced decision graph. `HOODGInducer` is a subclass of `OODGInducer` that provides a hill-climbing implementation of the algorithm.

6.4 NullInducer

`NullInducer` is a subclass of `Inducer` for which all calls, except the constructor, the destructor, and `description()`cause a `fatal_error`.

A Files

This appendix lists the files in which $\mathcal{MLC}$++ class declarations and definitions are located. The corresponding files are "[File Stem].h" and "[File Stem].c", using [File Stem] from the following table.

Class Name	**File Stem**
Array <>	Array
Array2 <>	Array2
ASCIIPref	DisplayPref
AttrCategorizer	AttrCat
AttrInfo	Attribute
AttrValue	Attribute
AugCategory	AugCategory
AugCategoryAlloc	AugCategory
BadCategorizer	BadCat
BagCounters	BagCounters
Categorizer	Categorizer
CatGraph	CatGraph
CatTestResult	CatTestResult
ConstCategorizer	ConstCat
ConstInducer	ConstInducer
CrossValidator	XValidator
CtrInducer	CtrInducer
DblLinkList <>	DblLinkList
DecisionTree	DecisionTree
DisplayPref	DisplayPref
DLLPix <>	DblLinkList
DotGraphPref	DisplayPref
DotPostscriptPref	DisplayPref
DTCategorizer	DTCategorizer
GenPix <>	GenPix
GLD	GLD
HOODGInducer	HOODGInducer
ID3Inducer	ID3Inducer
Inducer	Inducer
Instance	Instance
InstanceAlloc	Instance
InstanceInfo	InstanceInfo
LabelledCtrInstanceBag	CtrBag
LabelledCtrInstList	CtrInstList
LabelledInstance	Instance
LabelledInstanceAlloc	Instance

LabelledInstanceBag	BagSet
LabelledInstanceGenerator	LabInstGen
LabelledInstanceInfo	InstanceInfo
LabelledInstanceList	InstList
LearnCurve	LearnCurve
MLCIstream	MLCStream
MLCOstream	MLCStream
MOptions	MOptions
MRandom	MRandom
MString	MString
NominalAttrInfo	Attribute
NullInducer	NullInducer
OODGInducer	OODGInducer
PtrArray <>	Array
PtrArray2 <>	Array2
RDGCategorizer	RDGCat
RealAttrInfo	Attribute
RootedCatGraph	RootCatGraph
TDDTInducer	TDDTInducer
TableCategorizer	TableCat

References

[Koh94] Ron Kohavi. Bottom-up induction of oblivious, read-once decision graphs. In *Proceedings of the European Conference on Machine Learning*, April 1994. Paper available by anonymous ftp from starry.stanford.edu:pub/ronnyk/euroML94.ps.

[Qui86] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986. Reprinted in Shavlik and Dietterich (eds.) Readings in Machine Learning.

E N D