The Future of MLPACK

Ryan Curtin

September 1, 2010

Abstract

Currently, FASTLIB and MLPACK are poorly maintained quagmires of libraries that have been designed with a large number of suboptimal design decisions. While the implemented machine learning methods are fast and scalable, the API is ad-hoc and unorganized. This document inspects each of these suboptimal design decisions, suggests alternatives, and lays out a future plan to clean up and modernize the code. The resultant library will be called MLPACK (the FASTLIB name will be removed from use), will depend as much as possible on external components, will use modernized C++ coding standards (and stop relying heavily on preprocessor macros), will provide bindings to other common languages, and an automatic build server and benchmarking suite will be put in place. All future MLPACK development will stand on four principles outlined in this document.

Contents

1 Introduction

FASTLIB and MLPACK are scalable C++ machine learning libraries, meant as a machine learning analog to LAPACK, the well-known linear algebra library. MLPACK is the library of actual machine learning methods, whereas FASTLIB is an auxiliary library that MLPACK is built on top of. Development of these libraries is ongoing but neither FASTLIB nor MLPACK have reached a point where they could be released as stable, although preliminary versions have been released.

Current alternatives (or competitors) to FASTLIB and MLPACK include Weka3 Hall [2010], Java-ML Abeel [2009], Apache Mahout The Apache Foundation [2010], and the various available MATLAB tools to perform machine learning. The strength of MLPACK lies in it scalability and speed, though, and this is the point where its competitors fall short. Therefore, the main goal of MLPACK is to produce a fast, scalable machine learning library; that is, faster and more scalable (hopefully significantly) than other available machine learning tools.

In the past, FASTLIB and MLPACK were developed by four or five members of the FASTlab, but recently they have begun to graduate and move on to bigger, better things. Unfortunately, this left FASTLIB and MLPACK without a set of dedicated maintainers, and as a result the code is now being maintained by a few students who previously had no knowledge of the library and, upon entering the project, were entirely unfamiliar with the codebase and had to learn it by hand. It will be shown, however, that this is a good thing; a set of fresh eyes can help redesign the API in a more user-friendly manner.

To plot the course for the future of FASTLIB and MLPACK, we must first establish the long-term goals of the libraries. As mentioned earlier, scalability and speed are large priorities and will help establish MLPACK as a serious competitor to the other libraries out there. However, the library must also be accessible and easy to use. In addition to this, because the FASTlab is a machine learning research lab, MLPACK can (and should) provide implementations of cutting-edge machine learning algorithms that no other library has support for. Therefore, combining these thoughts, we can devise the following simple list of goals for MLPACK:

- Create scalable, fast machine learning algorithms
- Design an intuitive, simple API for users who are not C++ gurus
- Implement as large a collection as possible of machine learning methods
- Provide cutting-edge machine learning algorithms that no other library does

We must keep all four of these goals in mind when working on MLPACK.

2 Apparent Design Decisions

In the following sections we will take a close look at each of the design decisions made by the previous FASTLIB/MLPACK team and analyze each of them in-depth. Several important topics must be addressed; they are listed below.

- **Build system**: FASTLIB and MLPACK were written with a build system called `fl-build`, which requires developers to learn a specific in-house build file format, as well as requiring end users to modify environment variables and learn to use the system so that they can build and install the library.

- **C++ Standards**: The coding standards used by FASTLIB and MLPACK are mostly consistent and appear to follow Google's open-source standards Weinberger et al. [2010], but not completely. All of the standards that FASTLIB and MLPACK adhere to need to be discussed in-depth and revised; especially the gratuitious and unnecessary use of preprocessor macros.

- **Object Traversal Library**: In `fastlib/base/otrav.h` resides an object traversal library implemented entirely in templated functions written as preprocessor macros. This is an abhorrent abuse of C++, and while perhaps nifty, creates more confusion and problems than it actually solves. We will look at suitable alternatives.

- **Debugging Macros**: Like in `otrav.h`, FASTLIB defines preprocessor macros, many of which are used for debugging purposes and implement mechanisms for which standard implementations already exist. We will take a look at these macros and discuss how they will be replaced or removed entirely.

- **Unit Testing**: Also defined with preprocessor macros is a confusing unit testing library that is not actually utilized by much of the code. Other unit testing solutions already exist and have existed for a long time; these will be discussed. Additionally, the lack of coverage of the unit tests will be addressed.

- **`Vector` and `Matrix` Classes**: FASTLIB implements its own set of Matrix and Vector classes. In addition to having an unwieldy API, and again relying heavily on preprocessor macros, these classes are simple data structures for which external solutions have existed for years. FASTLIB also defines several classes, such as `ArrayList` and `Queue`, that are implemented (and have been implemented for years) in the Standard Template Library.

- **Dependencies on External Libraries**: Similar to the previous topic, FASTLIB implements all kinds of functionality that has been supported by external libraries for many years. Among these in-house creations are a CSV and ARFF reader, which are actually incomplete and buggy, as well as a text file parsing module. It is senseless to force ourselves to maintain code like this when we can depend on a project whose purpose is to make those functions work right. We will address alternatives to existing problems of this nature.

- **Namespaces and General API Structure**: Currently, FASTLIB and MLPACK only use namespaces in a couple places, and the uses are generally non-intuitive and confusing, from an end-user's point of view. In addition to this, object hierarchies and general API structure is not coherent across the library. We will look at the more confusing cases of this (there are many), and then discuss solutions to these serious problems.

- **Input Parameter System**: Again, similar to the previous point, FASTLIB implements its own input parameter system, written in C in `fastlib/fx/fx.c`. This system actually provides functionality not found in any other library, but we will investigate whether or not such functionality is really even necessary, and discuss alternatives.

At the end of our analysis, we will collect the changes we plan to make (as well as compact justifications) and list them in a simple format for those who don't care to read through them all.

3 Custom Build System

3.1 The Problem

The original designers of FASTLIB designed an entire build system called `fl-build`. This build system is written in Python, and defines a file format of build file where a developer specifies the files to be compiled for a particular module and the dependencies of that module. Usage of the system by an end-user might involve them typing something like `fl-build build tree` or similar; however, that is only after the user has set several environment variables and added the `fl-build` executable to their path.

The entire build system resembles very closely the build system internal to Google, and the syntax for the `build.py` build files is particularly similar to that of Google's system. However, many of the intricacies and niceties of Google's system are entirely absent, and as a result leaves the average user wondering why `fl-build` was implemented in the first place.

The average end user, if downloading an application from source, will expect to be able to issue the following commands to build any package:

```
$ ./configure
$ make
# make install
```

Many will become irritated and look for another solution if the process turns out to be more difficult in that.

The `fl-build` system does not work well with package managers. In fact, it will attempt to download packages and compile them from source (specifically Trilinos for sparse matrix support), and link against those versions. It counter-intuitively keeps knowledge of the libraries it links against in hidden files located somewhere inside the build tree, and this is not easily documented to the user.

All in all, the `fl-build` system is an incredibly confusing piece of work that serves more to drive potential users away than to alleviate the problems associated with building a large library. The problem of build systems has been solved for many years. GNU Autotools The GNU Project [2010], CMake Kitware [2010], and SCons Knight [2010] are three readily available, standard build system. Autotools is the most standard of these systems for Linux-based packages, but it is not cross-platform (neither is `fl-build`).

3.2 The Solution

The solution settled on for FASTLIB and MLPACK is to use CMake, which is a cross-platform build system used for many high-profile applications and libraries such as Boost, KDE, MiKTeX, and OGRE (to name a few). While it has its

own unique build file format that MLPACK developers will have to learn and adopt, it must be noted that every single build system has its own language, and CMake's is significantly easier and more extensible than using Autotools.

With CMake, a user can configure and install MLPACK very simply:

```
$ cmake ./
$ make
# make install
```

A developer can use the out-of-source build functionality to keep two configured versions of the code simultaneously. For instance:

```
$ mkdir build_debug
$ cd build_debug
$ cmake -DDEBUG ../
$ make
```

and then concurrently

```
$ mkdir build_release
$ cd build_release
$ cmake -DRELEASE ../
$ make
```

and then the developer can test the speed of the library with the release-configured build, and debug issues with the debug-configured build. For more information on the concept of an 'out-of-source build', see `http://trac.research.cc.gatech.edu/fastlab/wiki/UsingCMake` where it is explained in much more detail. Overall, CMake provides all of the functionality that `fl-build` does (and more), although sometimes in a slightly different manner than `fl-build` does. In addition, CMake has a large community of developers and maintainers, so we do not have to rely on ourselves to fix any underlying bugs.

3.3 Current Status

The transition to CMake has already been done and completed, as of April 2010. A few bugs remain to be worked out, but the library will build just fine. One main difference is that the library builds as one component, not a bunch of small pieces. Therefore, if something is modified in `mlpack/allnn`, everything in `mlpack/` must be compiled to regenerate `libmlpack.a` (or whatever name the build artifact is given). While this is a slight drawback, it ensures to the end user that all the components of MLPACK are in one place. Resultantly, it does not suffer from the same problem Trilinos does, where there are a plethora of individual components with names that have seemingly nothing to do with what the functionality actually is, and the user has to read complex documentation to figure out which pieces they want to use.

While the transition to CMake was completed already, only a few have started to actually use CMake, as `fl-build` was left in the subversion trunk repository to give a transitional period to graduate students who, for the most part, have more important things to be doing. It is likely that the only way to actually effect a transition to everyone using CMake, once and for all, will be to simply remove the `fl-build` system and force CMake upon everyone.

4 C++ Style Concerns

Currently, FASTLIB and MLPACK are written in a certain style of C++ that resembles the Google open source C++ standards Weinberger et al. [2010], but not entirely. This may be a result of many of the core FASTLIB developers having interned at Google in previous summers (in fact, one works there full-time now). The main points that are currently adhered to are as follows:

- Constructors do nothing; instead, `Init()` is called to initialize an object
- Copy constructors are disabled to prevent accidental copying of objects
- Operator overloading is avoided in most cases (there are a few exceptions)
- Class hierarchies with virtual functions are mostly avoided
- Namespaces are not utilized often, if at all (there are a few exceptions)
- Functions that modify an object take a pointer to the object in question (pass-by-pointer)

- Preprocessor macros are used almost everywhere
- `friend` classes and functions are used frequently for testing purposes (as well as, in some cases, non-testing purposes)
- Exceptions are never used
- Most functions are preceded by a block comment that describes its functionality, but comments are generally sparse in the code
- 80-character line width, 2-character tabs

Each of these points (and later, other concerns) will be addressed individually. It is important to remember the goals that were established earlier in our discussions; out of the four goals we stated, the most important in this case is creating code that is simple and intuitive to use.

4.1 Constructors

A comment found in `mlpack/allnn/allnn.h` accurately sums up the current FASTLIB ideology for constructors[1].

```
// Default constructors should be kept very simple and should never
// allocate memory.  Their two responsibilities are to ensure that
// it's safe to destroy the object without having otherwise used it
// (e.g. to set pointers to NULL) and to poison memory when in debug
// mode with BIG_BAD_NUMBER = 2146666666 = NaN as a double and
// BIG_BAD_POINTER = 0xdeadbeef.
```

The use of a separate `Init()` function solves a couple of problems a user might encounter. Firstly, the constructor cannot return any error values, unless it throws an exception. An `Init()` function allows the return of a success or failure value. However, a developer must then worry about whether an object has had `Init()` called on it. On the other hand, a user does not necessarily need to know the parameters of an object at its creation time, if it uses the `Init()`-style initialization. This is not a very compelling point, however, since the user can just call the constructor again or set the then-unknown parameters to their desired values using auxiliary functions.

Currently, we will use the FASTLIB ArrayList class as an example. This class offers three functions:

- `ArrayList::Init()` - initialize an empty ArrayList
- `ArrayList::Init(index_t size, index_t cap = size)` - initialize an ArrayList with size `size` and optionally a capacity
- `ArrayList::Init(index_t size)` - initialize an ArrayList with a given size

In addition, there are thirteen other `Init`-like functions (such as `InitCopy()` and `InitSteal()` and all sorts of others) a developer has to know about to use ArrayList efficiently.

Therefore, a piece of sample code (albeit useless sample code) might look like this:

```
ArrayList<int> a;
a.Init(5); // five objects, we don't know them yet
for(int i = 0; i < 5; i++)
  a[i] = MethodThatReturnsANumber(i);

// now make a copy of a
ArrayList<int> b;
b.InitCopy(a);
```

Now suppose that instead of `InitCopy()`, we wrote a copy constructor that accepted a reference to another `ArrayList`, and copied each of the elements over. For syntatic elegance we'll also overload the `=` operator to use this copy constructor. Also suppose we define a constructor that takes a size as a parameter. Now, our code becomes simpler:

```
ArrayList<int> a(5); // allocate memory for five objects
for(int i = 0; i < 5; i++)
  a[i] = MethodThatReturnsANumber(i);

// now make a copy of a
ArrayList<int> b = a;
```

[1] The use of macros will not be addressed here, but later on.

We've removed two lines, and in addition to this, by overloading the constructor for each of these cases, we do not require the user to learn thirteen different named functions to initialize an `ArrayList`. It is important to note that in this case we have not defined exactly how we will overload our constructor thirteen ways and retain some semblance of syntactic simplicity; however, we will not address these concerns now, but in a later section.

Before we come to a conclusion, though, we have not addressed the fact that constructors cannot return a success indicator. However, a comprehensive look through the entire FASTLIB and MLPACK codebase reveals that out of 952 instances of an `Init()`-like function, only five of these (yes, only five) return anything other than `void`, and only one of those functions is meant to be called by the user (`DatasetInfo::InitFromFile()`). In addition to this, 230 of these `Init()`-like functions are not actually called `Init()`, giving the developer all sorts of headaches trying to remember the API.

Therefore, we can see that the advantages of using an `Init()` function, in the absolute vast majority of cases pertaining to FASTLIB, aren't even leveraged. Code simplicity and maintainability can be increased by using constructors, and on top of that, developers will have an easier time using the library. The user will not have to worry about whether or not their objects have had `Init()` called on them, and this will also help eliminate functions that were written twice, for both initialized and uninitialized parameters (this is prevalent in `fastlib/la/la.h`).

4.2 Copy Constructors

In most classes in FASTLIB, copy constructors are disabled via the use of the `FORBID_ACCIDENTAL_COPIES()` macro. The macro itself merely defines the copy constructor and = operator as private. Below is the code and documentation:

```
/**
 * Disables copy construction and assignment.
 *
 * This will save you needless headaches by preventing accidental
 * copying of objects via forgetting to use references in function
 * arguments, etc.  Instead of odd behavior you will get compiler
 * errors.  The FASTlib style guide recommends disabling copy
 * construction and assignment for classes that do more than store
 * data, defining the Copy method in their lieu if appropriate.
 *
 * Always follow member declaration macros with the appropriate
 * visibility label (public, private, or protected).
 *
 * Example:
 * @code
 *   class MyClass {
 *     FORBID_ACCIDENTAL_COPIES(MyClass);
 *    private:
 *     ...
 *   };
 * @endcode
 */
#define FORBID_ACCIDENTAL_COPIES(C) \
   private: \
    C (const C &src); \
    const C &operator=(const C &src);
```

So, with the case of `ArrayList`, which we inspected earlier, we are forced to write two lines of code to copy, instead of just one. This is a minor inconvenience to developers. `std::vector` (which we will actually discuss later), for instance, allows the copy constructor in the same way one might expect `ArrayList` to.

Let us describe case of user error that the comment refers to. Suppose a mildly clueless developer writes a function to take the trace of a matrix and writes the function with the following signature:

```
double Trace(Matrix x, Matrix y) { ... }
```

When this function is called, because references were not used, the compiler will make an unnecessary copy of the `x` and `y` matrices. That is, the function is pass-by-value; `x` in the scope of the called function is merely a copy of the variable passed to it from a higher scope, which is created using the copy constructor. The correct signature (ignoring const-correctness) would be

```
double Trace(Matrix& x, Matrix& y) { ... }
```

and as a result no copies would be made. The `FORBID_ACCIDENTAL_COPY` macro will cause a compilation error in the first, incorrect signature, due to the copy constructor being declared as private, and as a result a user who has written the incorrect code will not spend hours and hours debugging it. On the other hand, when a user wants an actual copy of an object, they need to define a `Copy()` function in the class and use its unstandardized syntax, instead of just using the assignment operator.

The tradeoff here, then, is between code simplicity and error prevention. Allowing simple, elegant code with the use of copy constructors and overloaded operators also allows less thoughtful programmers to shoot themselves in the foot. However, consider the two possibilities and how they play out for our theoretical thoughtless programmer. With disabled copy constructors, he always encounters an error message upon compilation, and it is likely that he figured out how to fix it (though not necessarily understanding exactly why the protective mechanism of the private copy constructor is in place), and merely does that every time he encounters the error. However, supposing the buggy code compiled, and ran but produced odd results, he might spend hours debugging it and trying to figure out why his code did not act how it should. At the end of this long journey, when the solution was finally found, the lesson would be learned, much more effectively, and our theoretical thoughtless programmer would be slightly less thoughtless and be a more competent, knowledgeable C++ programmer. In short, cleaning up for users allows them to be lazy and does not force them to actually learn the language. With C++, a lack of understanding of this issue indicates a lack of understanding of what references are and how to use them, which is an incredibly fundamental concept.

4.3 Operator Overloading

Copy constructors and the overloaded assignment operator discussion we have just completed can be further extended to all operators. Let us take a look at the code a user currently has to write to perform some basic matrix multiplication ($\mathbf{y} = 2(\mathbf{A}\mathbf{B}^T)\mathbf{v} + 3$):

```
void doMath(const Matrix& A, const Matrix& B, const Vector& v) {
  Vector y; // our output vector
  Matrix C; // we will have to store intermediate information in C
  la::MulTransBInit(A, B, &C);
  la::MulInit(C, v, &y);
  la::Scale(y, 2);
  // no scalar add to vector in la::
  for(int i = 0; i < y.length(); i++)
    y[i] += 3;
}
```

Note that one fairly simple line of matrix and vector math takes seven lines of code, including one loop. In addition, because the `Vector` and `Matrix` class use `Init()`-style constructors, all of the linear algebra functionality in the `la::` namespace must be written for all the possible cases of initialized and uninitialized matrices and vectors. This gives the user a considerable number of extra functions to worry about, and all sorts of method name permutations to keep in mind.

Imagine if we defined the following operator overloads (and one auxiliary transpose function):

```
Matrix& Matrix::operator*(const Matrix& rhs);
Vector& Matrix::operator*(const Vector& rhs);
Vector& Vector::operator*(const double rhs);
Vector& Vector::operator+(const double rhs);
Matrix& trans(const Matrix& mat);
```

These are only a few functions out of the potentially overloadable functions. For now we won't consider how we would actually implement the `trans()` function efficiently (we will just assume it is done, and in a later section we will introduce a library that does this for us). Use of these operators will not leak memory[2].

Now, using these functions, our function becomes much simpler:

[2] I have been approached before by programmers who claim that operator overloading introduces unnecessary temporary variables, but this is untrue for properly written overloads. Remember, because we are using references, these copies will not be created. The matrix library we will introduce later on also uses a lazy evaluation technique that prevents the creation of temporaries through the use of recursive templates and template metaprogramming. However, we could avoid creation of temporaries without using delayed evaluation by merely overloading only the `*=`, `+=`, and other in-place operators.

```
void doMath(const Matrix& A, const Matrix& B, const Vector& v) {
  Vector y = 2 * (A * trans(B)) * v + 3;
}
```

Seven lines of code and calls have been reduce to one simple, elegant line that reads almost as easily as if written in MATLAB. In the interest of simplicity, this is the clear choice, as opposed to forcing developers to learn a plethora of differently-named functions in order to use the library. When we combine the earlier decision to remove `Init()` functions and use constructors, we no longer need to worry about passing an uninitialized matrix as a parameter and do not need to write separate functions for the cases where an uninitialized matrix is passed as a parameter.

4.4 Class Hierarchies

For the most part, FASTLIB and MLPACK avoid using class hierarchies. This is a smart decision in some ways; the use of virtual functions adds to execution time significantly, in some cases. Supposing that we were to create a hierarchy of `Metric` classes using a virtual `Distance()` function, our resultant execution time when we called that virtual distance function on every entry in a large matrix would be exorbitantly high (as compared to not using virtual functions). This is because virtual functions require an extra lookup, and when the function is called millions of times in a single execution, the price is high. However, it must be noted that the price is negligible for most other cases, where the function is not being called nearly as much.

Virtual inheritance can help promote code reuse; thereby increasing simplicity and maintainability. For instance, the `AllNN`, `AllkNN`, and `AllkFN` classes are not related at all, even though an inspection of their code reveals extreme similarity. Perhaps a better solution would be to have `AllkNN` and `AllkFN` inherit `AllNN` and then overload only the necessary methods; virtual inheritance (and the runtime cost that accompanies it) isn't even required in this case.

A very useful concept in class hierarchy design is that of policy-based design Alexandrescu [2001]. In this scheme, classes are templatized and the template parameters are 'policies' that influence how the class works. The best example for this (which is utilized occasionally in MLPACK) is that of a distance function. Suppose we define two distance functions as follows:

```
class ManhattanDistance {
  static double Distance(Vector& a, Vector& b) {
    return sum(abs(a - b));
  }
};

class UselessDistance {
  static double Distance(Vector& a, Vector& b) {
    return rand();
  }
};
```

Now, we define our templated machine learning method (we will use all-nearest-neighbors as an example) in such a way that allows the user to plug in whichever distance function they like:

```
template<
  class Distance = DefaultDistance
>
class AllNN {
  ...
  void CalculateNearestNeighbor(Vector& query_point) {
    ...
    for(int i = 0; i < num_points; i++) {
      distances[i] = Distance::Distance(query_point, points[i]);
    }
    ...
  }
  ...
};
```

In this fashion, a user can use any predefined distance classes or write their own (as long as it implements `Distance()`). We can define any number of policies for a specific class, and provide generic machine learning algorithms that a user can

easily configure to their very specific needs. An interested reader is directed to Alexandrescu [2001] for more reading on the capabilities of policy-based design.

Overall, MLPACK should use a combination of both inheritance (occasionally virtual inheritance, where speed concerns are not too important of an issue) and policy-based design to provide a simple, intuitive interface for users while still remaining as fast as possible.

4.5 Namespaces

Namespaces, a very important C++ feature, are rarely utilized by MLPACK (and the occasional uses do not make intuitive sense). In addition, MLPACK is not under any main namespace. We consider it self-evident that namespaces must be introduced. Everything should fall under a top-level `mlpack::` namespace, and namespaces below that should be chosen in an intuitive, organized manner. For instance, file-manipulation functions could be put into `mlpack::file::`, the all-nearest-neighbors method could be put into `mlpack::allnn::`, and so on.

4.6 Pass-by-pointer and Pass-by-reference

Functions in FASTLIB and MLPACK that modify an input parameter currently operate on the old C-style pass-by-pointer idiom:

```
void DoSomething(Matrix* willBeModified);
```

This is an old C-style concept, and actually will result in the pointer itself being copied (although this particular concern has no real performance impact). Instead, C++ should use references:

```
void DoSomething(Matrix& willBeModified);
```

While using references as parameters is not always necessarily possible, the general C++ rule is: use references where possible; use pointers only when necessary. MLPACK does not adhere to this, and therefore, because we are using C++ and not C, should be modified so that they do.

4.7 Preprocessor Macros

It is well known that preprocessor macros are considered evil (but sometimes unfortunately necessary) in almost every single C++ programmer's mind. However, this was apparently ignored in the implementation of FASTLIB! FASTLIB and MLPACK together contain more than 1500 preprocessor macros (one thousand, five hundred). This is entirely absurd and every attempt should be taken to remove these entirely. Many of these macros are in the object traversal library (`fastlib/base/otrav.h`) and will be discussed later, in that section.

In some cases, macros are unfortunately unavoidable, but the bizarre extent of their use in FASTLIB and MLPACK must be eliminated.

Two of the macros being used present a particular problem. These are the `likely()` and `unlikely()` macros, which actually depend on another macro, defined below (from `fastlib/base/compiler.h`):

```
#ifdef __GNUC__
#define expect(expr, value) (__builtin_expect((expr), (value)))
#else
#define expect(expr, value) (expr)
#endif

#define likely(cond)   expect(!!(cond), 1)
#define unlikely(cond) expect(!!(cond), 0)
```

These two macros appear everywhere in the codebase, but they are not a good idea. Firstly, profiling information can easily be used to perform the same function more effectively. A developer can take a guess at what is likely and unlikely, but at least some of the time, they will be incorrect, and this will result in a huge performance hit. Using a profiler will give empirical data to the compiler for how to optimize, instead of a potentially sleepless developer's possibly misguided input. On top of this, developers are wasting time by doing this process by hand when automated tools (a profiler, that is) will do it for them. These macros cause unnecessary code bloat with no real advantage. Therefore, we must stop using them altogether and let the profiler perform its intended function.

Another confusing and unnecessary macro is below, from `fastlib/base/common.h`:

```
#if defined(SCALE_MASSIVE)
#define LI L64
#elif defined(SCALE_LARGE)
#define LI "l"                  /* TODO: confirm correct. */
#elif defined(SCALE_NORMAL)
#define LI ""
#endif
```

The purpose of this macro is to define an identifier that can be used with `printf`-style functions so that the `index_t` class will always print correctly. However, `printf()` is C, not C++! With C++, we should be using the iostream library, where serialization of (native) types is handled for us, and this macro is entirely unnecessary.

On top of this, the `LI` macro exposes a major problem with macros: name collisions. The sparse matrix portion of FASTLIB uses a library called Trilinos (more on this in a later section), but Trilinos also defines an `LI` macro. The programmer who noticed this passed judgment on macros in his comment, but strangely did absolutely nothing to resolve the issue:

```
// trilinos uses this identifier, which is not so surprising.
// this is why you should think twice about macros.
#ifdef LI
#undef LI
#endif
```

In fact, by undefining the macro, including this file `fastlib/sparse/sparse_matrix.h` in your code breaks all kinds of other code that depends on the `LI` macro!

Some more confusing macros exist, that aren't even used in the code (`fastlib/base/common.h`):

```
/** Size of a kilobyte in bytes. */
#define KILOBYTE (((size_t)1) << 10)
/** Size of a megabyte in bytes. */
#define MEGABYTE (((size_t)1) << 20)
/** Size of a gigabyte in bytes. */
#define GIGABYTE (((size_t)1) << 30)
/* Add more of these as necessary. */
```

These macros are referenced nowhere (with the exception of `MEGABYTE`, which is used twice), which could lead us to believe that the programmer who wrote this file was intentionally adding preprocessor macros where they weren't even necessary, resulting in code bloat and unmaintainability. Preprocessor macros are some of the most unmaintainable code a programmer can produce, and on top of that, they can be notoriously difficult to debug.

A programmer new to FASTLIB and MLPACK might even think that the presence and ubiquitiousness of such a ridiculous number of preprocessor macros was an intentional malicious act purely for the twisted humor of the core developers of the library!

4.8 `friend` Classes and Functions

Use of `friend` classes and functions is not very prevalent, and certainly not as absurd as the use of preprocessor macros. It should be kept in mind, though, that member functions should be used where possible, and `friend` functions only when necessary. Classes meant to test another class are a perfect example of where `friend` functionality should be used (and this is generally how this concept is used in FASTLIB and MLPACK).

4.9 Exceptions

As discussed earlier in the section concerning constructors, there is no simple way to return an error value from a constructor other than throwing an exception. The inline operators also suffer from this problem. Many C++ programmers mistakenly avoid exceptions due to a fear that it incurs a high runtime cost. However, this is not so ISO/IEC [2006]; many exception-handling implementations incur no run-time cost unless an exception is thrown, and the executable size increase from the compiler-generated exception management code is not incredibly significant (less than 15% on compilers, as of four years ago – it is almost certainly better now).

However, exceptions do introduce an extra level of complexity into the code: exception safety. In the case of MLPACK, though, it would not make sense for the code to have failure transparency. In a machine learning method, an exception would only arise if something were seriously wrong, and because this is merely a machine learning method and not an

interactive user application, the program can (and most likely will) terminate. Intuitively, the majority of situations where an exception would arise in MLPACK would be due to either erroneous coding or bad input data; neither of these can be fixed at runtime, and it is reasonable to expect the method to fail and terminate.

As a result of our decision that methods will mainly terminate upon exceptions, we do not need to put the same worry into guaranteeing that all allocated memory is freed before exit. The operating system will handle the reclamation of that memory (although it is not best practices to rely on that, the alternative would be an incredible amount of extra work and complexity).

Therefore, in conjunction with our decision to use constructors and operator overloading, we will use exceptions to denote failures in MLPACK methods that cannot be recovered, and when an exception is thrown, generally, it will be propagated to the user in the form of an error message. This interface is about equivalent to the current error reporting system (which is written mainly using preprocessor macros).

4.10 Comments In Code

Below is an example of a method found in MLPACK (`fastlib/tree/kdtree.h`):

```
/**
  * Creates a KD tree from data, splitting on the midpoint.
  *
  * @experimental
  *
  * This requires you to pass in two unitialized ArrayLists which will contain
  * index mappings so you can account for the re-ordering of the matrix.
  * (By unitialized I mean don't call Init on it)
  *
  * @param matrix data where each column is a point, WHICH WILL BE RE-ORDERED
  * @param leaf_size the maximum points in a leaf
  * @param old_from_new pointer to an unitialized arraylist; it will map
  *        new indices to original
  * @param new_from_old pointer to an unitialized arraylist; it will map
  *        original indexes to new indices
  */
 template<typename TKdTree, typename T>
 TKdTree *MakeKdTreeMidpointSelective(GenMatrix<T>& matrix,
                                      const Vector& split_dimensions,
     index_t leaf_size,
     ArrayList<index_t> *old_from_new = NULL,
     ArrayList<index_t> *new_from_old = NULL) {
   TKdTree *node = new TKdTree();
   index_t *old_from_new_ptr;

   if (old_from_new) {
     old_from_new->Init(matrix.n_cols());

     for (index_t i = 0; i < matrix.n_cols(); i++) {
       (*old_from_new)[i] = i;
     }

     old_from_new_ptr = old_from_new->begin();
   } else {
     old_from_new_ptr = NULL;
   }

   node->Init(0, matrix.n_cols());
   node->bound().Init(split_dimensions.length());
   tree_kdtree_private::SelectFindBoundFromMatrix(matrix, split_dimensions,
       0, matrix.n_cols(), &node->bound());

   tree_kdtree_private::SelectSplitKdTreeMidpoint(matrix, split_dimensions,
```

```
        node, leaf_size, old_from_new_ptr);

    if (new_from_old) {
      new_from_old->Init(matrix.n_cols());
      for (index_t i = 0; i < matrix.n_cols(); i++) {
        (*new_from_old)[(*old_from_new)[i]] = i;
      }
    }
    return node;
  }
```

We see easily that the comment preceding the code is quite detailed and tells a user how to utilize the function and what each of the parameters are. This is great; however, there are absolutely no comments in the actual implementation of the function! This leaves the user (or a potential maintainer) in a difficult situation when trying to figure out exactly what it is the code does. With no comments, the user or maintainer might have to spend hours poring over the code to fully understand it.

A much better style is to have that incredibly descriptive block comment at the top, and then in the actual code, comments should be scattered throughout. Here is another example of how *not* to comment code (and also how *not* to utilize horizontal and vertical whitespace to make code easy to read) (`mlpack/allnn/main.cc`):

```
int main (int argc, char *argv[]) {
  fx_module *fx_root=fx_init(argc, argv, NULL);
  AllNN allnn;
  Matrix data_for_tree;
  std::string filename=fx_param_str_req(fx_root, "file");
  NOTIFY("Loading file...");
  data::Load(filename.c_str(), &data_for_tree);
  NOTIFY("File loaded...");
  allnn.Init(data_for_tree, fx_root);
  //GenVector<index_t> resulting_neighbors_tree;
  //GenVector<double> resulting_distances_tree;
  NOTIFY("Computing Neighbors...");
  allnn.ComputeNeighbors(NULL, NULL);
  NOTIFY("Neighbors Computed...");
  fx_done(fx_root);
}
```

At some point, MLPACK code will need to be commented in a more usable fashion, and all new code should be thoroughly commented, not just in a block comment at the top of the function.

4.11 Actual Code Formatting

The current MLPACK and FASTLIB code is formatted in a sane, coherent manner. Tabs are two spaces; maximum line width is 80 characters. This is reasonable, but with sometimes very long machine learning method names, 80 characters of width can be restrictive. Therefore, this constraint should be relaxed, and 160 characters is more reasonable (and is somewhat standard among projects for which 80 characters is too restrictive). However, this being a point of extreme contention for many, it should not be particularly strictly enforced.

5 Object Traversal Library

Inside of `fastlib/base/otrav.h` one can find an object traversal system implemented entirely in preprocessor macros. The ostensible goal of this system is to allow a user to print arbitrary data containers; the system also lists its other purposes as deep-copying and advanced memory management.

A user needs to include a number of macros inside of their class definition, and then they can use the functionality that the `ot::` functions provide. Let us inspect one of these preprocessor macros:

```
#define OT_ENUM_EXPERT(x, T, print_code) \
    if (true) { \
      ot__visitor->Name(#x, x); \
      switch (x) { \
        print_code \
       default: \
        ot__visitor->Enum(x); \
      } \
    } else NOP // require semicolon
```

Many of them are templated function definitions inside of a preprocessor macro:

```
#define OT_TRANSIENTS(C) \
   public: \
    template<typename TVisitor> \
    friend void OT__TraverseTransients(C *ot__obj, TVisitor *ot__visitor) { \
      ot__obj->OT__TraverseTransients_(ot__visitor); \
    } \
   private: \
    template<typename TVisitor> \
    void OT__TraverseTransients_(TVisitor *ot__visitor)
```

The documentation for these macros (not included in the snippets shown above) is complex and tedious, and requires significant reading on the part of the user to actually understand what is going on.

In total, there are 55 macros of this ilk defined in this file, and for a user to debug any of the number of problems these might cause, the user will need to understand most of them (because most of them depend on other macros).

As discussed earlier, macros are a terrible thing, and make debugging particularly difficult. For instance, if one tells the object traversal library to traverse an object that it actually cannot (like a `std::string`), it will cause a very nasty segfault, because the macro functions insert a hook into the destructor that cause difficult-to-debug errors. The errors make no mention of being related in any way to the object traversal library, and one actually has to compile with debugging symbols and step through the code to figure out that the object traversal library is responsible (even a backtrace will not help, since macros are replaced with their definitions at compile time).

Clearly, this library is a bulky, bizarre construct with all sorts of maintainability and readability issues. On top of that, libraries already exist that perform not only this functionality, but in a more standardized fashion. Boost.Serialization Ramey [2004], a component of the ubiquitous Boost C++ libraries Beman Dawes [2010], is an apt choice for a replacement. The Boost libraries provide all sorts of functionality that the standard C++ language libraries do not, and several other Boost components can (and will) be used in MLPACK. It is likely that many of the Boost libraries will be included in the upcoming C++0x standard.

Our solution, using Boost.Serialization, ignores the benefits of deep copy and advanced memory management that the object traversal library provides, but realistically, advanced memory management is not necessary, and users should be writing copy constructors that perform deep-copy correctly anyway.

Therefore, Boost.Serialization is a smart path to take as a replacement, and the process of replacement has already begun, as of Spring 2010.

6 Debugging Macros

FASTLIB contains an entire file full of macros that are intended for debugging (`fastlib/base/debug.h`). Generally, these macros involve printing output when verbose program options are specified, although macros exist for poisoning pointers and values. These macros tend to use other macros, including the `likely()` and `unlikely()` macros discussed earlier (which, as we noted, are terrible ideas). Nonetheless, the end goal of the verbosity macros is easily accomplishable without the use of macros, with a unified monolithic logging object. An object like this, with a smartly written `VerboseOut(...)` function (or something similarly named), will perform just as quickly as the existing macros, and as an object, will be able to contain state information, and is therefore more extensible and useful than the verbosity macros.

However, we have not addressed the number and pointer poisoning macros (`BIG_BAD_NUMBER` and `BIG_BAD_POINTER`). The intention of these macros is that a user declares values to default to those values, and then can easily see when they have not been properly initialized in a debugger. However, this is dependent on the user remembering to initialize their members to these values! In addition, uninitialized value problems are fairly easy to track down and solve with a debugger, without the need for poisoning. Given the extra code bloat that those macros cause when used in every class, from a maintainability perspective, these macros should not be bothered with.

In conclusion, a centralized singleton logging object is a more robust way to handle the needs of debugging output, and avoids the use of confusing macros. This object (or a similarly designed singleton) could additionally handle input parameters (this will be discussed in a later section).

7 Testing

The file `fastlib/base/test.h` defines a few macros which are intended to be used as a test suite. However, as even the comments say, it is experimental and not particularly robust.

The requirements for a test suite in terms of maintainability are slightly different. Generally, an MLPACK user will not be writing unit tests for their own development purposes; the people writing these tests will be MLPACK developers. Therefore, it is permissible to use a slightly more complex and site-specific solution.

One existing, robust solution is the Boost.Test framework Rozental [2007], which includes a unit testing framework. The Boost testing framework is somewhat similar to the existing prototypical system found in `test.h`. A developer defines their test functions, registers them with the UTF (unit testing framework), and the rest is automatically done. The features included in Boost.Test are numerous, though, and are all documented on the Boost website Rozental [2007]. Among the features that could potentially be used include an execution monitor, hierarchical test frameworks, and pre-written input parameter handling for the tests.

Unfortunately, much of the code inside of MLPACK is not properly unit-tested, and much is only tested at the machine learning method level; in addition, many of the machine learning methods have no implemented tests at all. Clearly, this is an issue that needs to be addressed. It would be reasonable to require the original author of each method to write suitable unit tests for all of their code.

8 `GenMatrix`, `GenVector`, and Related Classes

FASTLIB implements several already well-implemented classes: `GenMatrix`, `GenVector`, `ArrayList`, `Queue`, `MinHeap`, and a few others. We will investigate each of these classes individually.

8.1 `GenMatrix` and `GenVector`

The `GenMatrix` and `GenVector` classes (or the more commonly used typedef `Matrix` = `GenMatrix<double>` and `Vector` = `GenVector<double>`) are matrix and vector objects intended to provide a user with a fast class that interfaces easily with LAPACK and BLAS. As we noted earlier in this document, these classes do not use operator overloading and require an `Init()` function. Additionally, these classes introduce complex topics to the user, allowing the actual data pointer to be aliased by other Matrix objects. These functions are particularly confusing and require the user to think at a lower level of abstraction than is desirable when working at the higher level of machine learning algorithms.

Ideally, a matrix and vector class should provide the user with both syntactic elegance and maintainable simplicity. Very many libraries that provide both of these exist; a handful are listed below.

- Armadillo C++ Matrix library; `http://arma.sourceforge.net`
- Eigen C++ template linear algebra library; `http://eigen.tuxfamily.org`
- Boost.Math library; `http://boost.org`
- GNU Scientific Library (GSL); `http://www.gnu.org/software/gsl`
- Blitz++ scientific computing library; `http://www.oonumerics.org/blitz/`

These are only a few of the potential options that exist. Out of these options, both Eigen and Armadillo are template-based libraries that use template metaprogramming to perform lazy evaluation (which allows elegant operator overloading without the creation of unnecessary temporaries). We will discuss the Armadillo library due to its focus on large matrix support (Eigen does not specifically focus on this issue, which is a very important issue to us).

As we mentioned, the Armadillo library implements operator overloading with the use of lazy evaluation, which allows code like

```
Matrix Y = (A + B) * c; // A and B are matrices; c is a double
```

to be evaluted without the need for any temporaries. For more information, see the Armadillo documentation Sanderson [2010].

Let us reference the example we used in an earlier section, where we were calculating $\mathbf{y} = 2(\mathbf{A}\mathbf{B}^T)\mathbf{v} + 3$:

```
void doMath(const Matrix& A, const Matrix& B, const Vector& v) {
  Vector y; // our output vector
  Matrix C; // we will have to store intermediate information in C
  la::MulTransBInit(A, B, &C);
  la::MulInit(C, v, &y);
  la::Scale(y, 2);
  // no scalar add to vector in la::
  for(int i = 0; i < y.length(); i++)
    y[i] += 3;
}
```

We can see how clearly complex this is. Were we to be using the Armadillo library, the method would be:

```
void doMath(const mat& A, const mat& B, const vec& v) {
  vec y = 2 * (A * trans(B)) * v + 3;
}
```

This is immeasurably more intuitive, simple, and maintainable than the code that uses `GenMatrix` and `GenVector`! There is no need for either of those classes when such a suitable alternative exists.

Speed concerns are certainly an issue for these two classes. However, various benchmarks[3] of both simple and complex operations using both libraries show that speed differences are negligible, and at times, the Armadillo library outperforms the FASTLIB implementation. It should also be kept in mind that both implementations use LAPACK, which helps account for the speed similarities.

In addition to all of the above-mentioned advantages, Armadillo provides one more advantage over the FASTLIB implementation: it is maintained and actively developed. The FASTLIB implementation has not been seriously modified for years, whereas the Armadillo project releases new versions of their library almost once a month, with speed improvements as well as expanded functionality.

Therefore, FASTLIB will have the `GenMatrix` and `GenVector` classes entirely removed, and they will be replaced by the Armadillo equivalents. Many of the utility linear algebra functions found in `fastlib/la/la.h` are implemented already in Armadillo, and if not, it is trivial to adapt the old algorithms to operate on Armadillo matrices and vectors. This task is already in progress in the `fastlib-stl` branch.

8.2 `ArrayList`

The `ArrayList` class found in `base/col/arraylist.h` is a resizable array, much like the ArrayList class found in Java. However, something like this already exists in the C++ STL — `std::vector`. An inspection of the methods provided and implementation of `ArrayList` shows that `std::vector` provides everything that `ArrayList` does, with more standardized, elegant syntax and no serious performance hit.

An inspection of the uses of this class reveal that it is not meant to be used in places where performance is critical (a fixed-size array serves that purpose significantly more efficiently), although in places it is used in that fashion (in the all-nearest-neighbors, all-k-nearest-neighbors, and all-k-furthest-neighbors methods in MLPACK, to name a few).

Therefore, replacement of `ArrayList` with `std::vector` is sensible, and is currently being undertaken in the `fastlib-stl` branch.

8.3 `Queue` and `MinHeap`

There are a few other classes in FASTLIB that simply re-implement existing implementations of basic components that are found in the STL. Similarly to `ArrayList`, `Queue` should be replaced by `std::deque` and `MinHeap` is replacable by `std::priority_queue`. As it is not sensible to force FASTLIB maintainers to maintain components that are already implemented and maintained better than our own implementation, we will move to the STL equivalents. This work is being performed in the `fastlib-stl` branch, and will be merged back into `trunk/` upon completion.

8.4 `String`

Similar to the other few classes discussed above, the `String` class in `fastlib/col/col_string.h` should be replaced with `std::string`. The STL implementation provides a larger set of functionality, and (as with the other STL components) is standardized and familiar to most programmers.

[3] While not shown here, the specific results are available upon request by the interested reader.

9 Dependencies on External Libraries

The previous section concerned a problem prevalent all over FASTLIB and MLPACK: the library re-implements (sometimes poorly) methods and components which have been freely available via external libraries for a very long time. This causes all sorts of API and code bloat, and requires maintainers to spend their time fixing bugs in the home-grown implementation of these components. When depending on an external library, though, maintainers can simply pass bugs upstream. The upstream project is far more likely to have a correct, efficient implementation than the FASTLab coders are. After all, most students in the FASTLab are concerned with developing machine learning algorithms, not implementing support code.

The following is a list of components that should be replaced with external components. The best replacement has not always been found for each of these components, so replacements are not provided in this list.

- The CSV/ARFF reader found in `fastlib/data/`; this implementation is buggy and does not support all types of ARFF or CSV.
- File reading utilities in `fastlib/file/`; these implement functionality already mostly available in the STL, and with no more CSV/ARFF reader that depends on them, there is no reason to keep these.
- Threading support in `fastlib/par/` is immature, merely wraps pthreads, and should be replaced by something more robust.
- Many functions in `fastlib/math/` are already implemented in standard math libraries (`math.h`) and can be removed.
- The sparse matrix implementation that currently depends on Trilinos should be replaced (`fastlib/sparse/`); Trilinos is not the best solution for this problem (one potential option is implementing sparse matrix support for the Armadillo project).
- The input system in `fastlib/fx/` should be replaced by something more standard (more on this topic later).

Overall, it must be remembered that MLPACK should strive **only** to implement machine learning methods, and nothing else that it does not absolutely need. This is sound from a maintainability perspective, because MLPACK maintainers will not need to be concerned with fixing bugs in support classes but only bugs in the actual machine learning methods. In addition, the probability of bugs existing in the code decreases with the amount of code.

10 Name of the library

Before we continue with this discussion, we must note that in the FASTLIB and MLPACK literature it is not agreed upon what the library is called, how it is capitalized, or what the components are. General consensus holds that FASTLIB is the library that is the set of base functionality on top of which MLPACK (the machine learning methods) are built; however, different people will give differing views on this.

We note that in previous sections we have called for the removal of many primary components of FASTLIB; therefore, it makes sense to rename the library to simply MLPACK. This will help prevent user confusion (as well as developer confusion). From this point onwards, we will use MLPACK to refer to the entire library.

11 Namespaces and General API Structure

Currently, the namespace scheme in MLPACK is unstructured and not sensible. The following is a list of namespaces (and quick descriptions of what is contained therein):

- `data::` - Miscellaneous dataset-related routines
- `la::` - Linear algebra methods
- `learntrees::` - Regular pointer-style trees (as opposed to THOR trees)
- `linalg__private::` - Linear algebra utilities
- `math::` - Math routines
- `mem::` - Wrappers and tools for low-level memory management
- `optim::` - Here you can find all the classes you need for optimization

- `ot::` - Object traversal utilities for printing, copying, freezing, and serialization
- `proximity::` - Regular pointer-style trees (as opposed to THOR trees)
- `rpc::` - An asynchronous message-passing system
- `thor::` - THOR tree algorithm parallelization framework
- `tree::` - Regular pointer-style trees (as opposed to THOR trees)

This is not a very sensible structure; it is ad-hoc and not unified. In the earlier section we noted that the library should now be named MLPACK; following this, everything inside of MLPACK should lie in an `mlpack` namespace. Previously, there had been no base namespace, which could potentially cause a namespace conflict for users (although unlikely).

Several of the namespaces listed above can simply be removed, as we discussed the removal of those components earlier. These are `data::`, `mem::`, `ot::`, `la::`, `linalg__private::`, and potentially `math::`. Some of the methods within `la::`, `linalg__private::`, and `math::` do not exist in other libraries, so a `math::` namespace should be kept around for those instances.

In addition, none of the machine learning methods were encapsulated in their own namespace; this is a change which should be made.

Therefore, we will have a namespace structure which resembles:

- `mlpack::tree::` - all non-parallel tree data structures
- `mlpack::thor::` - tree parallelization framework
- `mlpack::thor::rpc::` - message passing framework for THOR
- `mlpack::optim::` - optimization-related classes
- `mlpack::allnn::` - allnn MLPACK method
- `mlpack::allknn::` - allknn MLPACK method
- `mlpack::allkfn::` - allkfn MLPACK method
- ...

This is not the final, definitive structure, and does not encompass everything, but individual components of the library should be encapsulated into specific, (if necessary) hierarchical namespaces.

12 Input Parameter System

MLPACK currently has a homegrown input parameter system in `fastlib/fx/fx.h`. One of the features of this system is that it allows hierarchical parameters; this allows for a "base set" of parameters common to all MLPACK methods to be provided. However, the developer must be familiar with the complex API that must be used. The following is an example of structure a user must define to specify the input parameters:

```
const fx_entry_doc infomax_ica_main_entries[] = {
  {"input_data", FX_REQUIRED, FX_STR, NULL,
   "The name of the file containing the mixture dataset (CSV or ARFF).\n"},
  {"lambda", FX_PARAM, FX_DOUBLE, NULL,
   "The learning rate (default 0.001)\n"},
  {"B", FX_PARAM, FX_INT, NULL,
   "Infomax data window size (default 5)\n"},
  {"epsilon", FX_PARAM, FX_DOUBLE, NULL,
   "Infomax algorithm stop threshold (default 0.001)\n"},
  FX_ENTRY_DOC_DONE
};

const fx_module_doc infomax_ica_main_doc = {
  infomax_ica_main_entries, NULL,
  "This performs ICA decomposition on a given dataset using the Infomax
method.\n"
};
```

The required structures can be more complex. While they are well documented, the code is unwieldy and there are simpler alternatives. Once those structures are defined, the user must initialize the system with:

```
fx_module *root = fx_init(argc, argv, &infomax_ica_main_doc);
```

Then, a user can access the actual values of the parameters with the `fx_param_*` set of functions. After the program is finished, the `fx_done()` function must be called.

This is not the entire functionality of the FX system; it also allows timers and outputs to be specified; when the program terminates, all of these timers and outputs are printed.

The FX system bears serious resemblance to the proprietary input parameter system designed and used at Google (which is not to be discussed outside of Google) although it is not as complete as that system. Certainly, continuing to use this system would be a bad legal idea, should the code ever come to inspection by any set of eyes that knew or cared. Legal issues with improperly licensed code do arise; one only needs to consider the plethora of patents Microsoft claims to hold over the Linux networking stack, or the various cases where GPL code has been improperly used (by TomTom in 2004, D-Link in 2006, and so forth).

Instead of this complex system that requires users to learn its counterintuitive interface, we should be using something more standard. Since `getopt()` (and `getopt_long()`) are not cross-platform (and also because they are ancient and have their own complexities), we should use Boost.ProgramOptions Prus [2004]. Using Boost components matches some previous design decisions we have made, and on top of that, gives a simple, powerful interface that we can use. Like the `fx` system, help dialogs are automatically generated. In addition, the Boost libraries are well-documented, unlike the current system.

13 Other Concerns

13.1 Automatic Build System and Benchmarking

Currently, there exists no streamlined way for the development team to know the functionality and performance of MLPACK. On top of this, many SVN commits break the build, leaving the code uncompilable; the programmers causing these problems are not notified (though they should have tested this before checkin!).

Many systems exist for this sort of task; an automatic build server like CruiseControl, CC.NET, Apache Continuum, or any number of other options would work well for this. These automatic build servers wait until a new revision of the code is submitted, and then build various configurations of the code. Additionally, automatic benchmarking could be set up in this process.

While it would be a tedious and time-consuming task to set this service up, it could be a very useful tool for analyzing where speed gains can be found in MLPACK (and for comparing against other machine learning libraries).

13.2 Bindings to Other Languages

While MLPACK is written in C++, many machine learning researchers do not possess a working knowledge of C++ and instead prefer to utilize a different language, like MATLAB or Python. Therefore, it is a smart idea to provide bindings to MLPACK for these users. SWIG Beazley [2010] can be used for automated Python binding generation. However, no similar tool exists for MATLAB. One can integrate C++ and MATLAB using MEX files, but each of these MEX files needs to be written by hand. In addition to these two popular languages, bindings can be made for R with the use of the Rcpp Eddelbuettel [2010] package.

We have provided three languages which we can generate bindings for; however, this is not an exclusive list, but only the languages most likely to be used. This goal is a priority, but not a top-level priority.

13.3 Compatibility Issues

Not all machine learning researchers are using Linux. Currently, MLPACK is written for Linux only, although it is for the most part platform independent and can be compiled on Mac OS X and Windows. However, only parts of the library work with Mac OS X. Therefore, it is a future goal of MLPACK to be compilable and compatible with most major operating systems, including mainly Linux, Mac OS, and Windows. As a result, the automatic build server (referenced in an earlier section) will need to have either virtual machines or physical hosts running different OSes to farm out to compile on.

14 Conclusion

We have discussed a variety of particular design decisions that will improve MLPACK in many ways and will set a good (albeit ambitious) course for future development and changes. The following section gives an itemized list of all of the

mentioned decisions for those who chose not to read the whole discussion. Again, these decisions are all made to follow these four goals:

- Create scalable, fast machine learning algorithms
- Design an intuitive, simple API for users who are not C++ gurus
- Implement as large a collection as possible of machine learning methods
- Provide cutting-edge machine learning algorithms that no other library does

14.1 Summary of New Design Decisions

Below is a list of all the design decisions detailed in this document. For the justification for each decision, refer to the section where it was discussed.

- **Use CMake for a build system**. This has already been done as of Spring 2010.
- **Modify C++ standards**.

  Do not use `Init()` functions, instead preferring constructors.

  Allow copy constructors.

  Use operator overloading to simplify algebraic operations.

  Use policy-based design to simplify class design (i.e. for distance functions)

  Namespaces should be used for everything, for organizational purposes.

  Use pass-by-reference wherever possible. Ensure `const`-correctness to prevent confusion.

  Stop using absurd macros.

  Avoid use of `friend` classes and functions where possible.

  Use exceptions to report errors; exception safety is not necessary as most will be fatal.

  Comment all code, not just function definitions.
- **Remove object traversal library**. It is unmaintainable and a mess.
- **Use a singleton object for debugging output, verbose output, and input parameters**. Avoid the use of macros, providing instead a more robust solution.
- **Use Boost.Test unit testing framework** since the existing test macros are not particularly robust or maintainable.
- **Stop using homegrown solutions**. They are unmaintainable and suboptimal.

  Use Armadillo C++ matrix library instead of `GenVector` and `GenMatrix`.

  Use `std::vector` instead of `ArrayList`.

  Use `std::deque` instead of `Queue`.

  Use `std::priority_queue` instead of `MinHeap`.

  Always use STL implementations before a homegrown solution.
- **Depend on external libraries** for non-machine-learning-specific functionality, but avoid unwieldy and large dependencies.

  Remove CSV/ARFF reader and use an external library.

  Remove file reading utilities and use functionality in the STL.

  Stop using `String` and use `std::string`.

  More robust threading support based on external libraries.

  Depend on external math libraries where possible (`math.h` in places).

  Replace sparse matrix implementation because Trilinos is the wrong solution.

  Remove input parameter system.

- **The library is now called MLPACK** because the FASTLIB/MLPACK duality is confusing to anyone not in the lab.
- **Use Boost.ProgramOptions for input parameters** and integrate with the earlier-mentioned input/output singleton object.
- **Set up automatic build and benchmark server** which will test each revision of the code.
- **Give bindings to other languages** such as MATLAB, Python, and R.
- **Assure compatibility with OSes** including mainly Linux, Windows, and Mac OS.

With all of these changes, it is clear that there is a lot of work to be done, but once they are all complete, MLPACK will be a much better library for both developers and users. If we stick to the design philosophies detailed above, the four goals we originally set out to accomplish should be simple and attainable.

References

Thomas Abeel. Java machine learning library (java-ml), August 2009. URL `http://java-ml.sourceforge.net`.

Andrei Alexandrescu. *Modern C++ Design*. Addison-Wesley Professional, 1 edition, 2001.

Dave Beazley. Simplified Wrapper and Interface Generator, June 2010. URL `http://www.swig.org/`.

Rene Rivera Beman Dawes, David Abrahams. Boost C++ Libraries, September 2010. URL `http://www.boost.org`.

Dirk Eddelbuettel. Rcpp: Seamless R and C++ Integration, July 2010. URL `http://dirk.eddelbuettel.com/code/rcpp.html`.

Mark Hall. Weka 3 - data mining with open source machine learning software in java, August 2010. URL `http://www.cs.waikato.ac.nz/ml/weka/`.

ISO/IEC. Technical Report on C++ Performance. February 2006.

Kitware. Cmake - cross platform make, August 2010. URL `http://www.cmake.org`.

Steven Knight. Scons: A software construction tool, August 2010. URL `http://www.scons.org`.

Vladimir Prus. Boost.ProgramOptions, 2004. URL `http://www.boost.org/doc/html/program_options.html`.

Robert Ramey. Serialization Overview, November 2004. URL `http://www.boost.org/doc/libs/release/libs/serialization/`.

Gennadiy Rozental. Boost Test Library, 2007. URL `http://www.boost.org/doc/libs/release/libs/test/`.

Conrad Sanderson. Armadillo: C++ linear algebra library, September 2010. URL `http://arma.sourceforge.net/`.

The Apache Foundation. Apache Mahout, April 2010. URL `http://lucene.apache.org/mahout/`.

The GNU Project. Autoconf, September 2010. URL `http://www.gnu.org/software/autoconf/`.

Benjy Weinberger, Craig Silverstein, Gregory Eitzmann, Mark Mentovai, and Tashana Landray. Google c++ style guide, September 2010. URL `http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml`.