mlpack:
A Vision for an Efficient Prototype-to-Deployment Machine Learning Library

March 18, 2021

For well over a decade, mlpack has provided efficient implementations of machine learning algorithms in C++ for use both by industry practitioners and researchers. In recent years, modern trends in the data science software ecosystem have focused on ease of prototyping, and this has resulted in a significant problem: the complexities of the Python ecosystem mean that deployment of Python-based data science workflows is often a laborious and complicated task. This problem also exists with other popular frameworks used for machine learning. In part, this is why the job data engineer or machine learning engineer exists. But mlpack only requires a modern C++ compiler, meaning deployment is easy, lightweight, and flexible. This guides our vision for the future: we can focus on developing mlpack into a library where both prototyping and deployment are easy, thus alleviating lots of the pain associated with deployment or productization of typical modern machine learning workflows.

This document gives a quick history of how mlpack got to where it is today (or at least at the time of this writing), then discusses the problems with current machine learning tools, and finally provides details and a high-level roadmap for making this vision a reality. All of this roadmap is pointed at making the following statement true:

The mlpack community is dedicated to demonstrating the advantages and simplicity of a native-C++ data science workflow.

1 A Quick History

Over the past few decades, the field of machine learning has seen an explosion of interest and excitement, with hundreds or even thousands of algorithms developed for different tasks every year. But a primary problem faced by the field is the ability to scale to larger and larger data—since it is known that training on larger datasets typically produces better results [34]. Therefore, the development of new algorithms for the continued growth of the field depends largely on good tooling and libraries that enable researchers and practitioners to quickly prototype and develop solutions [60]. Simultaneously, useful libraries must also be efficient and well-implemented. Up to this point, these reasons have been the motivation for our development of mlpack in C++.

mlpack was originally created in 2007 as “FASTLib/MLPACK”: a way to showcase the dual-tree algorithms [31] that were being developed at Georgia Tech. At that time, Python was not yet a popular choice for data science (and the term ‘data science’ was not in particularly widespread usage), and scikits.learn (as it was called at that time) [44] had just been created as a Google
Summer of Code project. C and C++ were popular languages to use for machine learning algorithms: the Shogun project [61], Shark [37], and Elefant [62] were all implemented in C++, and libraries like LIBLINEAR [30] and LIBSVM [8] chose C as their implementation language. It was only in the subsequent years, with the advent of Jupyter notebooks [40] and other tools that aided usability, that Python became the dominant language for machine learning.

Despite the rise of Python, C++ implementations provided by mlpack still regularly and significantly outperform competitors [15, 16, 19, 2, 28]. This led to mlpack positioning itself in 2010 as a library providing a wide variety of efficient, low-overhead implementations of machine learning algorithms—both standard and cutting-edge algorithms [10]. The choice was also made to depend on the Armadillo C++ linear algebra library, due to its support for compile-time optimization via template metaprogramming [56].

Indeed, in the years that followed, many new machine learning algorithms (often tree-based) were prototyped and/or implemented as part of mlpack: minimum spanning tree computation [41], fast max-kernel search [24, 23], density estimation trees [49], rank-approximate nearest neighbor search [50], furthest neighbor search [21, 17], kernel density estimation [32], $k$-means [45, 13, 29, 35], and other improvements for tree-based algorithms [22, 12, 11, 51]. Those years also led to several other development efforts, including:

- an automatic benchmarking system [28]: https://github.com/mlpack/benchmarks,
- an optimization framework now known as ensmellen [14, 20, 6]: https://github.com/mlpack/ensmallen,
- a visualization toolkit, mlboard: https://github.com/mlpack/mlboard,
- a repository of examples: https://github.com/mlpack/examples,
- a repository of pre-built models: https://github.com/mlpack/models,
- a Probot-based Github bot: https://github.com/mlpack/mlpack-bot, and
- continuous integration configuration: https://github.com/mlpack/jenkins-conf,

as well as several other efforts internal to the mlpack codebase (a collaborative filtering toolkit [2], a neural network implementation, a reinforcement learning toolkit, an ONNX converter, automatic bindings for other languages, and so forth). More about mlpack’s development practices and goals during this era can be found in a few documents from that period [27, 18, 10].

However, it has now been over a decade since mlpack’s design goals have been visited in detail, and the machine learning software world has changed significantly.
2 The Current State of Data Science

As machine learning has grown in importance, it is increasingly applied to problems in various domains; one example is the recent success shown by AlphaFold for protein structure prediction [57]. Other examples include applications in earth sciences [47, 59], telecommunications [9], and even agriculture [39]. The diversity of problems to which machine learning can be applied is truly huge; some fascinating examples even include plasma wave classification onboard satellites [64], exoplanet identification [58, 25, 46], and automated lunar rockfall mapping [7].

Resulting from this widespread applicability, an enormous collection of machine learning libraries are now available, including well-known libraries such as scikit-learn [44], PyTorch [43], and TensorFlow [1]. Numerous support libraries are also available; to name a few just in the field of natural language processing, HuggingFace’s transformers [63], SpaCy [36], and gensim [55] are widely used. Of course, Python is not the only ecosystem in which machine learning is done: the R project [48] and the Julia project [5] are popular alternatives. Importantly, all of these tools and ecosystems focus on making it very simple to prototype and tune a machine learning model, often inside of an interactive notebook environment such as that provided by the Jupyter project [40].

But a successful application of machine learning to a particular problem generally involves not just modeling, but the entire data science process shown in Figure 2. Most mainstream machine learning

![Figure 2: The typical prototype-to-deployment lifecycle for data science. Typical tools (bottom row) focus only on certain steps of the data science pipeline, meaning that multiple tools must be used together to cover every step of the machine learning lifecycle. mlpack (top row), in C++, can cover all of these steps with minimal overhead and complexity. In the future, we can improve mlpack’s support for the deployment process, resulting in mlpack being able to provide a feature-complete prototype and deployment pipeline.](image-url)
libraries, including the examples above, focus specifically on steps 2 through 6 (data ingestion through model evaluation), with limited facilities for deployment. When it is time to deploy the prototype—for instance, to a standalone server or an embedded device—machine learning code inside of a Jupyter notebook is not easily deployable.

Often, the process of deployment is handled by the researcher handing over their notebook prototype to a data engineer or machine learning engineer, whose job it is to deploy the model [33]. Sometimes, the entire prototype may need to be rewritten in a different language or using different tools more amenable to the deployment environment. While some tools in the Python ecosystem support deployment tasks, e.g., compiling TensorFlow programs to specific devices [38], this process is often tedious and may require systems knowledge that the original researcher who assembled the prototype may not possess.

This has led to a machine learning software landscape where prototyping and deployment are decoupled, with different packages focusing on each individual task. That leads to an inefficient situation where translation may be required between different toolkits when going from prototyping to deployment\(^1\). The bottom row of Figure 2 shows the steps of the machine learning pipeline that existing popular machine learning tools are focused on.

One of the primary reasons that the transition from prototype to deployment can be so difficult—especially when deploying to embedded devices—is the complexity of the environments typically used for machine learning prototypes. Python, the most popular language for data science tasks [52], has significant overhead, and a deployment target would need to have a Python interpreter and all dependencies available (this could require over a gigabyte of storage, plus noticeable memory overhead at runtime). This is a serious issue for resource-constrained environments, where a full-fledged Python interpreter may not be feasible. In addition, dependency management may become a significant issue, as it may be difficult to reconcile the required versions of dependencies for the prototype and the available versions of dependencies in the deployment environment. Often, the prototype may even be packaged in a heavyweight Docker container [42], which is of course not an option in an embedded setting and is inefficient even when it is feasible.

However, virtually every possible deployment environment has one thing in common: a C++ compiler can easily compile device-specific code with no need for the overhead of an interpreter. Thus, if the data science prototype was originally written in C++, the difficult parts of deployment would be alleviated: the C++ code could be compiled for use on any device without the need for significant changes. Dependency issues could be avoided via static linking (if needed).

\(^1\)!It’s hard to find an easy citation or hard data, but buy a drink for any industry data scientist and they’re highly likely to have stories of machine learning projects that failed because converting the prototype to deployment couldn’t be done, usually because the operations side of the company did not have tooling or expertise to adapt and deploy a notebook-format Python model, and the data science side was unable to provide something the operations side could work with.
3 A Vision For The Future

mlpack is uniquely positioned to solve the typical deployment difficulties of the previous section. Today, it is an efficient, easy-to-use C++ machine learning library. But with a little bit of modification and planning, mlpack’s development could aim towards realizing a better vision than what’s available today: a unified C++ development environment for data science prototyping and deployment, which can help remove lots of the pain felt both in industry and academia when trying to move from a proof-of-concept to a product.

Removing the deployment barrier is a matter of building upon the decades of work that have already been invested into the common set of tools used for low-level systems work, while retaining the fast prototyping interface that is so important for data science.

So, prototyping and development can be done in a familiar environment for data scientists via the use of xeus-cling interactive C++ Jupyter notebooks. These C++ notebooks are capable of the same plotting and exploration functionality that is so important for prototyping, and due to the nature of C++ being compiled, will generally be more performant than an equivalent Python notebook. An example of a C++ notebook with mlpack code can be seen in Figure 3.

Then, this exact C++ code can be easily exported from the notebook and directly compiled for deployment. This workflow is suitable for both large-scale deployments on powerful servers,

---

2See https://github.com/jupyter-xeus/xeus-cling.

Figure 3: Prototyping: a screenshot of mlpack usage in an interactive xeus-cling notebook. The notebook functions just like a user would expect a Python notebook to function; cells can be run, then modified and re-run, just as expected.
and low-power applications where minimizing computational overhead matters. Not only that, the workflow is simple—there is no need to spend significant amounts of time consulting deployment documentation that is specific to a toolkit or a library (for instance, TensorFlow’s XLA [38] is a great example of a very complex but very specific system for model deployment), because a user can simply invoke a standard set of tools (e.g., a properly-configured C++ compiler). An example of this is shown in Figure 4. Given the ubiquity of C++, it is even straightforward to embed C++ code inside of applications written in other languages, as most programming languages have packages to wrap C++ libraries [4, 26].

Next, we lay out a high-level plan for turning the vision into reality. We highlight nine different high-level areas that will need work inside and outside of mlpack to successfully realize a prototype-to-production pipeline with mlpack in C++. Here, we try to keep the goals high-level; individual details and status for each goal can be found on Github (https://github.com/mlpack/mlpack/) or the mlpack website (https://www.mlpack.org/).

1. **Interactive notebook prototyping support.** Via the xeus-cling project, users can easily develop their models in the same way that they might with a standard Python notebook using Python tools. This environment is interactive via the incremental cling compiler, and so it can give quick feedback to users. We already have notebooks deployed on the mlpack homepage at https://www.mlpack.org/, but there is still more to do: for instance, we could integrate notebook kernels more closely with our documentation, so that wherever a code snippet is displayed on our website, there can also be a ‘Run’ button to test it.

2. **GPU/accelerator support via bandicoot.** Modern machine learning applications are quite computationally intensive and therefore often make use of accelerator devices such as GPUs. Bandicoot, under development at https://gitlab.com/conradsnicta/bandicoot-code/, is meant to provide an API-equivalent alternative to Armadillo where computation is performed on CUDA or OpenCL devices. We should help get Bandicoot release-ready, and once that is done, the use of templates by mlpack and ensmallen should allow easy substitution of Bandicoot types for Armadillo types—but there will likely be some refactoring necessary. Given the speedups Armadillo regularly shows over other CPU-based linear algebra toolkits due in part to its template metaprogramming infrastructure, we can likely expect that when using Bandicoot under mlpack or ensmallen, we can see significant speedups over other GPU-based machine learning toolkits.
3. **Adaptable examples and documentation.** A library is only as good as its documentation, and therefore we should aim to demonstrate many real-world usages of mlpack that users can base their own code off of. As a bonus, examples are easy to benchmark and compare with other libraries, showing the speedups inherent in the C++-based approach. We can expand on the examples already available in the examples repository (https://www.github.com/mlpack/examples/), and even include example usages of mlpack in other languages. It would even be useful to include some examples of ensmallen usage without mlpack.

4. **Improved compilation time and memory usage.** A big problem with using mlpack today is that it requires a lot of resources when compiling (often, 4GB+ of memory is used to compile the tests). Part of the reason for this is heavy usage of boost and complicated design patterns like the visitor paradigm [3]. We should replace these, preferring instead to use simple dependencies and simple design patterns where possible. For instance, in many places, visitors can be replaced with virtual inheritance. Another tool for reducing memory usage and compilation time is to profile the compilation process; this should help expose the parts of mlpack that are difficult for the compiler, and we can then decide how to simplify them. It should (hopefully) be a reasonable goal to say that compiling an mlpack program should not take more than 1GB of memory.

5. **Better support for cross-compilation and lightweight deployment.** Users may be deploying mlpack in a wide variety of situations, and we want to have good support for all of these settings. Thus, we should update our CMake configuration and documentation to make it easy for users to, e.g., compile an mlpack program for a Raspberry Pi or any other embedded device. This also probably means bolstering our support for statically compiling programs that use mlpack, and reducing the size of the compiled code. Goal (4) above is a step in this direction, but we can go further than that and also make mlpack header-only to ease deployment significantly. (Note that Armadillo depends on a BLAS/LAPACK library, which will not be header only, so users will still have to do some linking, but it should only be against BLAS/LAPACK.) We can even include examples in our examples repository of how to deploy to a low-resource device.

6. **Utilities for non-numeric data.** Most machine learning algorithms are derived to operate on some matrix containing real numeric data, but most real-world data is not made up on only numeric data. This is often handled by conversion and encoding; for instance, many ecosystems have the concept of a ‘dataframe’ that allows loading arbitrary data types and then seamlessly passing them off as numeric data to machine learning algorithms. We have some support for this already via data::Load(), which can provide a data::DatasetInfo that automatically maps string data to integers. But a more capable dataframe class might be useful; one example in C++ is xframe, from the XTensor package (see https://github.com/xtensor-stack/xframe/). We could extend our support to provide something like xframe, or we could adapt mlpack so that it can work with xframe dataframes. In any case, it is an important goal that working with non-numeric data in mlpack feels straightforward, as it does in many other machine learning ecosystems.

7. **Pre-trained state-of-the-art models.** We currently have the models repository at https://github.com/mlpack/models/. This repository contains ready-to-use implementations of complicated models such as Darknet [54] and YOLO [53], so that users who want to deploy this type of model don’t need to handwrite it. It could be very useful to include more types of popular mod-
els (for instance, BERT and variants), and even provide pretrained weights and datasets for these models, so that users don’t need to go through the computationally expensive training process.

8. **Automatically-generated bindings**: unfortunately, no matter how nice of an interface we provide, there are many people who (justifiably) think that C++ is too complicated and difficult a language to use. We can meet these users where they are, by providing an interface in their language of choice. Currently, we have an automatic binding generation system that provides bindings of mlpack methods to Python, R, Julia, the command-line, and Go (and there is a PR open for Java support). There are several ways in which these bindings could be improved: the interface provided to other languages often does not feel ‘native’—typically, a single function is provided that can do training, prediction or both, as opposed to providing a class with functions in it. Also, mlpack models in other languages are typically black boxes: if you train a model in Python, you cannot access its weights from Python. Support could be added for model introspection in other languages, perhaps via JSON serialization in cereal. In addition, while it is quite easy to convert a Python usage of mlpack directly into, e.g., a Julia usage, it is a little less clear how to convert that Python usage to C++ for easy deployment. So we can also improve the documentation for that process, or even provide utilities that can automatically convert mlpack calls in other languages to C++.

9. **Efficient implementations**—both in terms of runtime and memory. This was the original goal of mlpack, and it still applies today: provide efficient implementations of machine learning algorithms. The tools and techniques to do this haven’t changed; profilers can be used to identify slow sections of code and improve them, and low-level implementation tricks can be used to get additional speed or throughput (SIMD instructions, OpenMP parallelization, etc.). We should continue to aim to improve the speed of our implementations; this allows us to produce benchmarks comparing mlpack favorably to alternatives, which in turn makes the argument for a C++-native data science approach even stronger.

Of course, that is not a comprehensive list, and it’s easy to think of other high-level development goals that could help us execute on this vision. For those goals we have listed, though, each of the following sections elaborate and detail individual bullet points. While previously, mlpack was focused primarily on providing efficient implementations of machine learning algorithms as kind of a C++ ‘swiss army knife’, we can now change our focus to making the following statement ring true:

*The mlpack community is dedicated to demonstrating the advantages and simplicity of a native-C++ data science workflow.*
References


