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

(a) 2007–2014. (b) 2014–2016. (c) 2016–current.

Figure 1: Logos over the years.

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 `ensmallen` [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

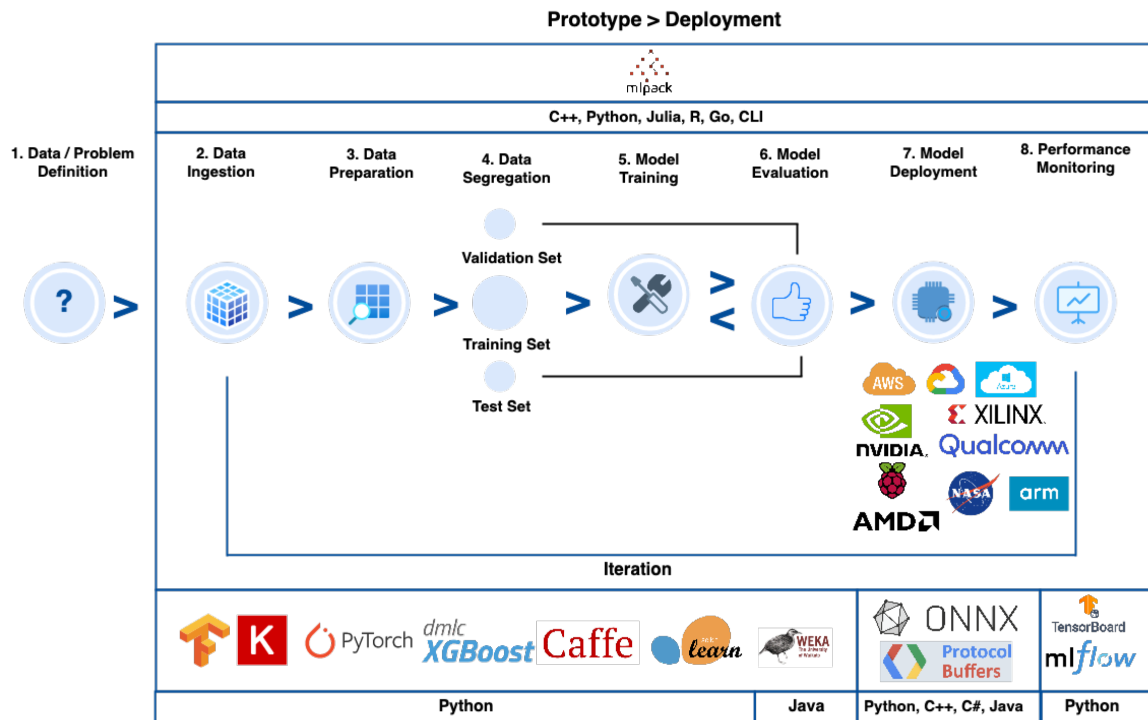

Figure 2: *The typical prototype-to-deployment lifecycle for data science. Typical tools (bottom row) focus only on certain steps of the data science pipeline, meaning that multiple tools must be used together to cover every step of the machine learning lifecycle. mlpack (top row), in C++, can cover 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.*

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[2]. 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,

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

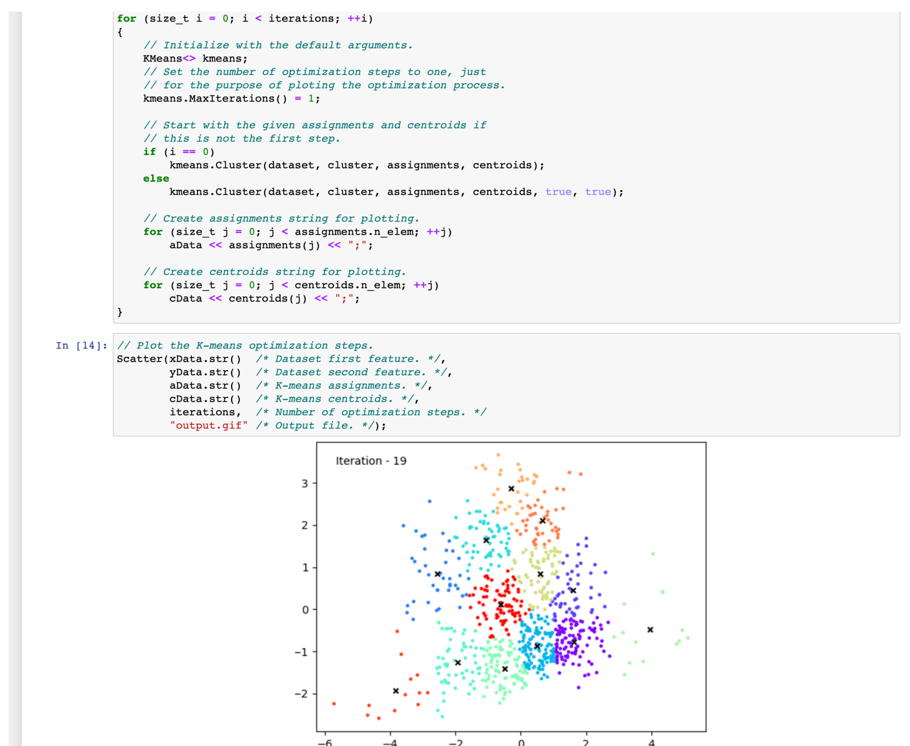

Figure 3: **Prototyping***: a screenshot of mlpack usage in an interactive* `xeus-cling` *notebook. The notebook functions just like a user would expect a Python notebook to function; cells can be run, then modified and re-run, just as expected.*

```
(: ryan@firfir :: ~ :) 47
(: 11:52:28 :) $ g++ -O3 -o mnist_simple mnist_simple.cpp -lmlpack -larmadillo -fopenmp
(: ryan@firfir :: ~ :) 48
(: 11:53:18 :) $ ./mnist_simple
Start training ...
Epoch 1
0.324228[====================================================================================================] 100% - ETA: 0s - loss: 0.323748
844/844 [====================================================================================================] 100% - 12s 14ms/step - loss: 0.324228
Validation loss: 916.66.
Epoch 2
0.133294[====================================================================================================] 100% - ETA: 0s - loss: 0.133097
844/844 [====================================================================================================] 100% - 12s 14ms/step - loss: 0.133294
Validation loss: 718.788.
Epoch 3
0.0986352===================================================================================================] 100% - ETA: 0s - loss: 0.0984893
844/844 [====================================================================================================] 100% - 12s 14ms/step - loss: 0.0986352
Validation loss: 752.628.
Epoch 4
616/844 [=========================================================================>...........................] 72% - ETA: 3s - loss: 0.08426423
```

Figure 4: **Deployment**: *compilation and (truncated) run of the* `mnist_simple` *mlpack example. The* `./mnist_simple` *binary could be easily deployed now; it would be easy to statically link or cross-compile depending on the needs of the deployment. This example code simply trains the model, but it would be easy to write a program that operates as a server that waits for test data, then issues a prediction.*

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

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D.G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] S. Agrawal, R.R. Curtin, S.K. Ghaisas, and M.R. Gupta. Collaborative filtering via matrix decomposition in mlpack. In *Proceedings of the ICML 2015 Workshop on Machine Learning Open Source Software*, 2015.

[3] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., USA, 2001.

[4] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31–39, March 2011.

[5] J. Bezanson, A. Edelman, S. Karpinski, and V.B. Shah. Julia: A fresh approach to numerical computing, 2015.

[6] S. Bhardwaj, R.R. Curtin, M. Edel, Y. Mentekidis, and C. Sanderson. ensmallen: a flexible C++ library for efficient function optimization. In *Proceedings of the Workshop on Systems and ML and Open Source Software at NeurIPS 2018*, 2018.

[7] V. T. Bickel, C. Lanaras, A. Manconi, S. Loew, and U. Mall. Automated detection of lunar rockfalls using a convolutional neural network. *IEEE Transactions on Geoscience and Remote Sensing*, 57(6):3501–3511, 2019.

[8] C.-C. Chang and C.-J. Lin. LIBSVM: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):1–27, 2011.

[9] C. Cui, R. Arian, S. Guha, N. Peyghambarian, Q. Zhuang, and Z. Zhang. Wave-function engineering for spectrally uncorrelated biphotons in the telecommunication band based on a machine-learning framework. *Physical Review Applied*, 12(3):034059, 2019.

[10] R.R. Curtin. The Future of MLPACK. `https://www.ratml.org/misc/mlpack_future.pdf`.

[11] R.R. Curtin. Faster dual-tree traversal for nearest neighbor search. In *International Conference on Similarity Search and Applications*, pages 77–89. Springer, 2015.

[12] R.R. Curtin. *Improving dual-tree algorithms*. PhD thesis, Georgia Institute of Technology, 8 2015.

[13] R.R. Curtin. A dual-tree algorithm for fast k-means clustering with large k. In *Proceedings of the 2017 SIAM International Conference on Data Mining*, pages 300–308. SIAM, 2017.

[14] R.R. Curtin, S. Bhardwaj, M. Edel, and Y. Mentekidis. A generic and fast C++ optimization framework. *arXiv CoRR*, abs/1711.06581, 2017.

[15] R.R Curtin, J.R. Cline, N.P. Slagle, M.L. Amidon, and A.G. Gray. MLPACK: A Scalable C++ Machine Learning Library. In *BigLearning: Algorithms, Systems, and Tools for Learning at Scale*, 2011.

[16] R.R. Curtin, J.R. Cline, N.P. Slagle, W.B. March, P. Ram, N.A. Mehta, and A.G. Gray. MLPACK: A scalable C++ machine learning library. *Journal of Machine Learning Research*, 14(Mar):801–805, 2013.

[17] R.R. Curtin, J. Echauz, and A.B. Gardner. Exploiting the structure of furthest neighbor search for fast approximate results. *Information Systems*, 80:124 – 135, 2019.

[18] R.R. Curtin and M. Edel. Designing and building the mlpack open-source machine learning library, 2017.

[19] R.R. Curtin, M. Edel, M. Lozhnikov, Y. Mentekidis, S. Ghaisas, and S. Zhang. mlpack 3: a fast, flexible machine learning library. *Journal of Open Source Software*, 3(26):726, 2018.

[20] R.R. Curtin, M. Edel, R.G. Prabhu, S. Basak, Z. Lou, and C. Sanderson. Flexible numerical optimization with ensmallen, 2020.

[21] R.R. Curtin and A.B. Gardner. Fast approximate furthest neighbors with data-dependent candidate selection. In *Similarity Search and Applications*, pages 221–235, 2016.

[22] R.R. Curtin, D. Lee, W. B. March, and P. Ram. Plug-and-play dual-tree algorithm runtime analysis. *Journal of Machine Learning Research*, 16(101):3269–3297, 2015.

[23] R.R. Curtin and P. Ram. Dual-tree fast exact max-kernel search. *Statistical Analysis and Data Mining*, 7(4):229–253, 2014.

[24] R.R. Curtin, P. Ram, and A.G. Gray. Fast exact max-kernel search, 2013.

[25] A. Dattilo, A. Vanderburg, C. J. Shallue, A. W. Mayo, P. Berlind, A. Bieryla, M. L. Calkins, G. A. Esquerdo, M. E. Everett, S. B. Howell, D. W. Latham, N. J. Scott, and L. Yu. Identifying exoplanets with deep learning. II. two new super-earths uncovered by a neural network in k2 data. *The Astronomical Journal*, 157(5):169, apr 2019.

[26] D. Eddelbuettel and J.J. Balamuta. Extending `R` with `C++`: A Brief Introduction to `Rcpp`. *The American Statistician*, 72(1):28–36, 2018.

[27] M. Edel. mlpack open-source machine learning library and community. *NIPS 2018 MLOSS Workshop*, 2018.

[28] M. Edel, A. Soni, and R.R. Curtin. An automatic benchmarking system. In *NIPS 2014 Workshop on Software Engineering for Machine Learning*, 2014.

[29] C. Elkan. Using the triangle inequality to accelerate k-means. In *Proceedings of the 20th international conference on Machine Learning (ICML-03)*, pages 147–153, 2003.

[30] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *the Journal of machine Learning research*, 9:1871–1874, 2008.

[31] A.G. Gray and A.W. Moore. 'N-Body' problems in statistical learning. In *Advances in Neural Information Processing Systems 14 (NIPS 2001)*, volume 4, pages 521–527, 2001.

[32] A.G. Gray and A.W. Moore. Nonparametric density estimation: Toward computational tractability. In *SIAM International Conference on Data Mining (SDM)*, pages 203–211, 2003.

[33] DHI Group. Dice tech job report: the fastest growing hubs, roles and skills. 2020.

[34] A. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, March 2009.

[35] G. Hamerly. Making k-means even faster. In *Proceedings of the 2010 SIAM international conference on data mining*, pages 130–140. SIAM, 2010.

[36] M. Honnibal, I. Montani, S. Van Landeghem, and A. Boyd. spaCy: Industrial-strength Natural Language Processing in Python, 2020.

[37] C. Igel, V. Heidrich-Meisner, and T. Glasmachers. Shark. *Journal of Machine Learning Research*, 9(6), 2008.

[38] Alphabet Inc. XLA: Optimizing Compiler for Machine Learning. `https://www.tensorflow.org/xla`, January 2021.

[39] A. Kamilaris and F.X. Prenafeta-Boldú. Deep learning in agriculture: A survey. *Computers and Electronics in Agriculture*, 147:70–90, 2018.

[40] T. Kluyver, B. Ragan-Kelley, F. Pérez, B.E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J.B. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and Jupyter Development Team. Jupyter notebooks—a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents, and Agendas*, pages 87–90, 2016.

[41] W.B. March, P. Ram, and A.G. Gray. Fast Euclidean minimum spanning tree: algorithm, analysis, and applications. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '10)*, pages 603–612, 2010.

[42] J. Nickoloff. *Docker in action.* Manning Publications Co., 2016.

[43] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library, 2019.

[44] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[45] D. Pelleg and A.W. Moore. Accelerating exact k-means algorithms with geometric reasoning. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 277–281, 1999.

[46] R.V. Petrescu, R. Aversa, A. Apicella, and F.I. Petrescu. Nasa data used to discover eighth planet circling distant star. *Journal of Aircraft and Spacecraft Technology*, 2(1):19–30, 2018.

[47] M. Pourshamsi, M. Garcia, M. Lavalle, and H. Balzter. A machine-learning approach to polinsar and lidar data fusion for improved tropical forest canopy height estimation using nasa afrisar campaign data. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 11(10):3453–3463, 2018.

[48] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2020.

[49] P. Ram and A.G. Gray. Density estimation trees. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2011)*, pages 627–635. ACM, 2011.

[50] P. Ram, D. Lee, W.B. March, and A.G. Gray. Linear-time algorithms for pairwise statistical problems. In *Advances in Neural Information Processing Systems 22 (NIPS 2009)*, volume 23, 2009.

[51] P. Ram, D. Lee, W.B. March, and A.G. Gray. Linear-time algorithms for pairwise statistical problems. In *NIPS*, pages 1527–1535. Citeseer, 2009.

[52] S. Raschka, J. Patterson, and C. Nolet. Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information*, 11(4):193, 2020.

[53] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016.

[54] J. Redmon and A. Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7263–7271, 2017.

[55] R. Řehůřek and P. Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA.

[56] C. Sanderson and R.R. Curtin. Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software*, 1:1–2, 2016.

[57] A.W. Senior, R. Evans, J. Jumper, J. Kirkpatrick, L. Sifre, T. Green, C. Qin, A. Žídek, A.W.R. Nelson, and A. Bridgland. Improved protein structure prediction using potentials from deep learning. *Nature*, 577(7792):706–710, 2020.

[58] C. J. Shallue and A. Vanderburg. Identifying exoplanets with deep learning: A five-planet resonant chain around kepler-80 and an eighth planet around kepler-90. *The Astronomical Journal*, 155(2):94, jan 2018.

[59] A. W. Smith, I. J. Rae, Claire Forsyth, D. M. Oliveira, M. Freeman, and D. Jackson. Probabilistic forecasts of storm sudden commencements from interplanetary shocks using machine learning. *Social Work*, 18, 2020.

[60] S. Sonnenburg, M.L. Braun, C.S. Ong, S. Bengio, L. Bottou, G. Holmes, Y. LeCun, K.R. Müller, F. Pereira, C.E. Rasmussen, G. Rätsch, B. Schölkopf, A. Smola, P. Vincent, J. Weston, and R. Williamson. The Need for Open Source Software in Machine Learning. *Journal of Machine Learning Research*, 8:2443–2466, December 2007.

[61] S. Sonnenburg, G. Rätsch, S. Henschel, C. Widmer, J. Behr, A. Zien, F. de Bona, A. Binder, C. Gehl, and V. Franc. The SHOGUN machine learning toolbox. *The Journal of Machine Learning Research*, 11:1799–1802, 2010.

[62] C. Webers, K. Gawande, A. Smola, C.H. Teo, J.Q. Shi, J. Yu, J. McAuley, L. Song, Q. Le, and S. Guenter. Elefant release 0.4. 2009.

[63] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, et al. HuggingFace's Transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[64] S. Yagitani, T. Toda, I. Nagano, K. Hashimoto, T. Okada, H. Matsumoto, and M. Tsutsui. Neural network for plasma wave classification onboard satellite. In *ISAP 1996 - International Symposium on Antennas and Propagation*, volume 3, pages 721–724, January 1996.