An Automatic Benchmarking System

Marcus Edel
Freie Universität Berlin
Arnimallee 7, 14195 Berlin
marcus.edel@fu-berlin.de

Anand Soni
Indian Institute of Technology
Bombay, Powai, Mumbai 400 076
anand.soni@iitb.ac.in

Ryan R. Curtin
Georgia Institute of Technology
Atlanta, GA 30332
ryan@ratml.org

Abstract

It is standard to write unit tests to answer the question 'does my software work?', but it is not always common to answer questions related to performance such as 'is my software fast?'. Existing approaches to this problem are often manually invoked, limited in scope, or require a fairly large amount of maintenance. For our machine learning library **mlpack**, we have developed an automated benchmarking system which is integrated into the Jenkins continuous integration tool; this allows our developers to quickly compare the performance of their implementations with earlier revisions or other libraries' implementations. The system is flexible and easily configurable; thus it would be straightforward to deploy for other projects.

1 Introduction

For the successful development and maintenance of machine learning algorithms, benchmarking is of paramount importance. A developer should be able to easily know how their algorithm performs compared to either other implementations or previous revisions of their own code, with respect to both runtime performance and other metrics (for a classifier, one example might be the classification accuracy). Despite this clear need, there exist relatively few attempts to address the issue [1]–[3]. Further, existing attempts are generally not very automated, meaning they have high maintenance requirements—and thus are of limited utility in the context of software development workflows.

We have developed an automatic benchmarking system for the needs of our machine learning library, **mlpack** [4]. Given some implementations of algorithms, sets of parameters with which to run each algorithm, and a set of datasets, the system will automatically gather benchmarking information for each combination of parameters and datasets, and store it in an SQLite3 database. This benchmarking information is not limited to runtimes, but can also include metrics such as precision and recall, like OpenML [3]. We have integrated our system into the Jenkins continuous integration tool [5]; this allows our system to operate in a relatively low-maintenance, turnkey fashion.

As we will show, the system is flexible and can be adapted to many different use cases. Below we list some of the capabilities of our system, which we use for `mlpack` development:

- Compare the runtimes of different libraries' implementations of the same algorithms for different datasets.
- Display the memory usage over time for a particular algorithm implementation.
- Show the changes in runtime for different revisions of a particular algorithm implementation for a particular dataset.
- Given some performance metric (i.e. precision, recall), compare the performance of different algorithms, or different implementations of the same algorithm.

The benchmarking results are stored in an SQLite3 database for low-effort retrieval of results. Due to the flexibility of SQL, the types of results that can be presented are limited only by the imagination of the user. We will present our own visualizations and results for **mlpack**, but it should be remembered that the system is quite flexible and its usage is not restricted to our setting. In fact, the core functionality of our system is not even restricted to machine learning applications.

Due to space constraints, we restrict ourselves to a fairly high-level overview here; however, more details on the project are available at `http://github.com/zoq/benchmarks/`.

2 System overview

We first clarify some terminology that we will be using. A **library** contains one or more **methods**, which are particular implementations of an algorithm. Each method may take different sets of **parameters** that configure their behavior. Lastly, **datasets** are the inputs to methods. Benchmarking often involves running methods with many different datasets and many different sets of parameters.

A primary obstacle for an automatic benchmarking system is varying interfaces for the methods being tested; for instance, methods may be written in different languages. In addition, the desired output—runtimes, performance, or other metrics—may be returned differently. This necessitates the construction of wrapper scripts to standardize the program interfaces. For each method to be benchmarked, our system expects a wrapper script implementing the following methods:

- `RunTiming(...)`: run the method on a particular dataset, collecting the runtime to be returned with `GetTime(...)`.
- `RunMetrics(...)`: run the method on a particular dataset, collecting performance information for a number of metrics.

Assuming that wrapper scripts have been written for each method of interest, the user must then specify a configuration file. This configuration file, in the YAML format (for easy readability), lists the methods and datasets for each library. Below is an excerpt, which specifies the datasets with which to run PCA for the **mlpack** library implementation. Three iterations are performed for each dataset, and timing measurements are collected.

```
library: mlpack
  methods:
    PCA:
      run: ['timing']
      iteration: 3
      script: methods/mlpack/pca.py
      datasets:
        - files: ['datasets/cities.csv', 'datasets/madelon.csv']
          options: '-d 2'
```

The configuration can be more complex, allowing selection of the metrics to be run as well as sets of parameters to use for particular datasets (or for all of the datasets).

Given a configuration file and corresponding wrapper scripts, a user can call `make CONFIG=config.yaml run` and the system will run all benchmarking targets. These results can be stored to a new database or appended to an existing one, for tracking of long-term trends.

2.1 Benchmarking algorithm runtime and memory usage

Simple measurements of runtime using wall clock time with a single run may suffer from inaccuracies. For instance, for programs written in Java, the JVM may not load classes until they are used. Thus a first run of the algorithm may be slower than subsequent runs. To care for these effects, the `iteration` configuration option may be specified for each method; this controls how many times the benchmarks are run for that method. In a typical configuration, a particular method will be evaluated four times, and the results from the first run would be removed from the final measurement.

In addition, to deal with extremely long-running methods, the user may also specify the `timeout` option. Dealing with more complex runtime issues may be handled inside of the wrapper script.

For our own usage with **mlpack**, the wrapper scripts that we use do not include the time it takes for data to load or save. This is because our interest is in benchmarking the actual method implementation and not also the efficiency of the dataset loading routines.

Another important measurement is memory usage over time. Our system can use the valgrind tool `massif` [6] for heap profiling, which records (among other things) the memory usage during the running of the program. Because of the large size and nature of the memory profiling results, the individual `massif .mout` files are stored individually on disk and the names of those files are stored in the SQLite database. The system will run memory profiling for each method in a configuration file with `make CONFIG=config.yaml memory`.

2.2 Benchmarking algorithm performance

Runtime and memory usage are not the only interesting measurements of a method; also interesting are other standard performance metrics such as precision, recall, or lift, to name a few. Thus, another important aspect of the benchmarking system is its ability to evaluate many performance metrics, in combination with a bootstrap analysis for measuring the variability of those metrics.

In order to perform an empirical evaluation the benchmark system implements different metrics[1] that can be used to analyze the parameter space and different variations. In addition, to permit averaging across metrics and datasets, the user may also specify to normalize the metrics to comparable scales. The benchmark system does this by scaling the performance for each dataset and metric from 0 to 1, where 0 is the baseline performance and 1 is the Bayes optimal.

One-dimensional performance measures, however, do not tell the full story because they do not take into account the intrinsic variability existing in the results. For instance, **mlpack**'s implementation of the perceptron classifier performs well on 10 out of 13 datasets and metrics, but the implementation performs poorly on small datasets. So what do these measures tell us about the behavior of the system in general? In order to answer this question, we implemented a bootstrap analysis, which computes the confidence intervals over the mean for the different metric estimations. The bootstrapping method is described as follows:

1. Randomly select a bootstrap sample (with replacement) from the available datasets.
2. Randomly select a bootstrap sample (with replacement) of the specified and implemented metrics for the bootstrap sample from Step 1.
3. Rank the specified methods by mean performance across the sampled dataset and metric.
4. Repeat Steps 1–3 for a fixed number of iterations. (This results in a number of potentially different rankings.)

Thus, the bootstrap analysis offers additional insight into the influence that a particular dataset or metric can have on the performance of a method.

3 Automatic integration with Jenkins

A software developer interested in a benchmarking system may want to answer many questions: Did my changeset cause speedup or slowdown? How does performance change with the number of training examples? How well does it handle high dimensions? Outliers or mislabeled data?

Ideally, the developer should be able to answer these questions quickly and without significant effort. Fortunately, due to the configuration file-based nature of our system, it is easy to integrate into an automated build process. Thus, we have fully integrated the system into the automated build process of the **mlpack** library, which uses the Jenkins continuous integration framework [5]. We have written wrapper scripts for methods from **mlpack**, WEKA [7], MATLAB [8], the Shogun Toolkit [9], mlpy [10], FLANN [11], ANN [12], and scikit-learn [13]. In our setup, a Jenkins job is provided for each library, which enables us to update our benchmarking results whenever a new release or change is made to any of the libraries. These jobs may be triggered either manually (for other libraries which require manual updates) or may be triggered automatically, by e.g. a commit to the source code repository. These new benchmarks are automatically inserted into our existing SQLite database.

4 Visual comparisons using d3

The benchmarking system is useless without a way to interpret the results. But with results in SQLite format, we can query the results in arbitrary ways. For our own purposes, we have used the JavaScript d3 library [14] to create a number of useful interactive visualizations. These are publicly accessible and regularly updated at `http://www.mlpack.org/benchmark.html`.

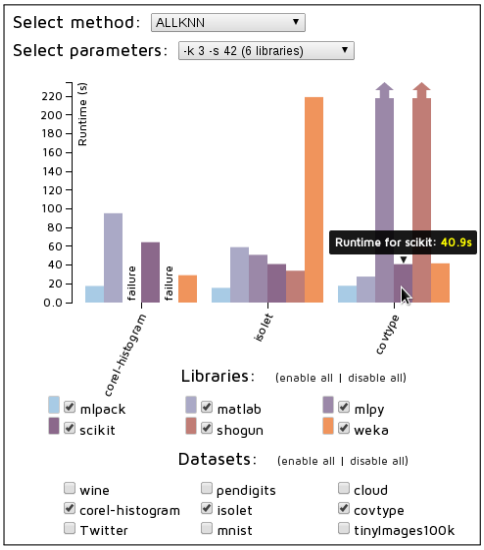
[1] See `https://github.com/zoq/benchmarks/wiki/`.

We have extracted some visualizations for demonstration here. Figure 1a shows a runtime comparison for the all-k-nearest-neighbors method implemented in various libraries for a couple of datasets. This view is useful for direct comparisons of methods from different libraries. Figure 1b shows runtimes for **mlpack**'s linear regression method on the `arcene` dataset for each successive release version of **mlpack**; this is very useful for regression testing of runtimes. For example, the graph shows a performance regression in January 2014, which was fixed in the following release (July 2014). Figure 1c shows performance metrics for two implementations of a perceptron with default parameters; in this case, scikit's implementation performs better for all metrics except precision.

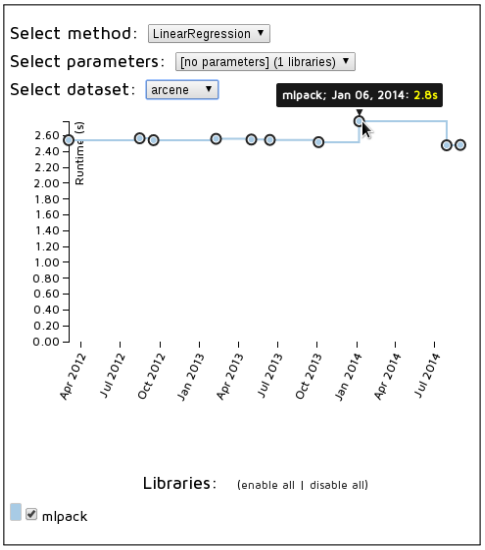
The possibilities for visualizations are nearly endless, and here we have not shown some other useful view ideas: both runtime and performance metrics can be compared across different methods entirely, across different sets of parameters for the same method, across different datasets, or any combination thereof. Thus, our system's result storage format offers significant flexibility for both typical and atypical benchmarking tasks.

5 Conclusion

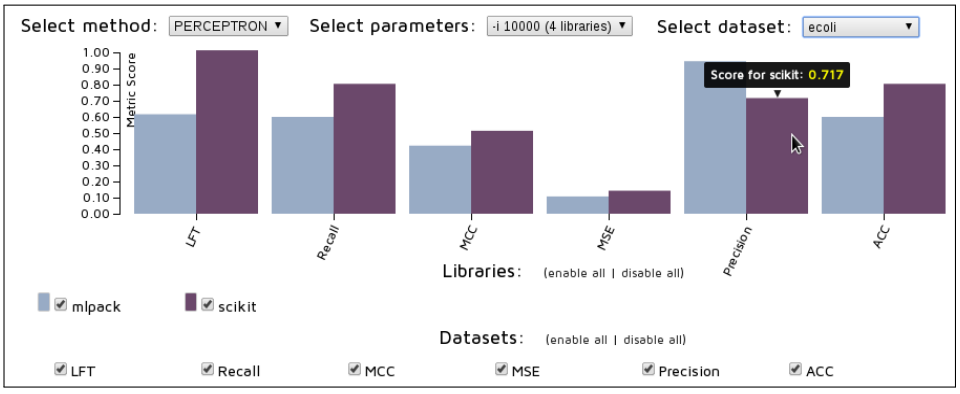
We have developed a flexible, extensible automatic benchmarking system for the development of **mlpack**, and integrated it with Jenkins. This allows our developers to quickly and easily know both how their own changes have affected the performance of the library, and how their implementations fare against other libraries. Further, the results of this system are shown on our website, helping prospective users to determine for themselves whether our library is the best choice for their needs.



(a) Runtime comparison d3 view.



(b) Historical comparison d3 view.



(c) Metric comparison d3 view.

Figure 1: Benchmark results views.

References

[1] J. Abernethy and P. Liang, *MLcomp*, Project homepage at `http://www.mlcomp.org/`, 2013.

[2] V. Niculae, *Scikit-learn-speed*, Project homepage at `http://scikit-learn.org/ml-benchmarks/`, 2013.

[3] J. N. van Rijn, B. Bischl, L. Torgo, B. Gao, V. Umaashankar, S. Fischer, P. Winter, B. Wiswedel, M. R. Berthold, and J. Vanschoren, "OpenML: a collaborative science platform," in *Machine Learning and Knowledge Discovery in Databases*, Springer, 2013, pp. 645–649.

[4] R. Curtin, J. Cline, N. Slagle, W. March, P. Ram, N. Mehta, and A. Gray, "MLPACK: a scalable C++ machine learning library," *Journal of Machine Learning Research*, vol. 14, pp. 801–805, 2013.

[5] K. Kawaguchi, *Jenkins*, Project homepage at `http://jenkins-ci.org/`, 2014.

[6] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan Notices*, ACM, vol. 42, 2007, pp. 89–100.

[7] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten, "The WEKA Data Mining Software: An Update," *SIGKDD Explorations*, vol. 11, no. 1, 2009.

[8] MATLAB, *version 8.1.0 (R2013a)*. Natick, Massachusetts: The MathWorks Inc., 2013.

[9] 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*, vol. 99, pp. 1799–1802, 2010.

[10] D. Albanese, R. Visintainer, S. Merler, S. Riccadonna, G. Jurman, and C. Furlanello, "mlpy: Machine Learning Python," 2012, Project homepage at `http://mlpy.fbk.eu/`. arXiv: `1202.6548 [cs]`.

[11] M. Muja, *Flann, fast library for approximate nearest neighbors*, Project homepage at `http://www.cs.ubc.ca/research/flann/`, 2011.

[12] S. A. David Mount, *Ann, approximate nearest neighbors*, Project homepage at `http://www.cs.umd.edu/~mount/ANN/`, 2010.

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

[14] M. Bostock, *D3.js - data-driven documents*, Project homepage at `http://d3js.org/`, 2012.