MyriaL Random Forests

(CSE 544 Research Project)

Connor Moore
Computer Science & Engineering
University of Washington
moorec22@cs.washington.edu

Nicholas Reiter
Computer Science & Engineering
University of Washington
nreiter@cs.washington.edu

1. PROBLEM AND BACKGROUND

As both datasets and the analytic demands upon them grow, the field of database management systems faces continued challenges in scalability and efficiency. One of the current common paradigms for improving both involves the development of shared-nothing database clusters. Piggybacking on the significantly decreased costs of commodity hardware, these systems can attain high levels of raw computing power, at the cost of increased management overhead and computing complexity. Over the years, a number of systems have become popularized, including Hive/Hadoop and Spark. Built to leverage the advantages of highly distributed systems, these systems typically allow easy scaling out with both dedicated and cloud clusters. More recent to the scene, the Myria Big Data Management Service, developed by researchers in the University of Washington, was designed to bring a number of improvements to these systems. Similarly built to run on scalable distributed and cloud-based systems, the Myria system contains a number of separate modules. This includes a cloud-based web interface; a Relational Algebra Compiler that allows code to be written in one of several languages, and compiled to a number of output languages; the MyriaX Parallel Data Management System, which can serve in place of Hadoop, Spark, and others; and the MyriaL serves as a relational algebra based language for the system.

Important to the adoption and usage of any new system are the quantity and quality of available libraries and source examples. Even technically superior systems have failed to attract developers in part due to lack of example and library code. As such, in order to both demonstrate feasibility, as well as contribute to the system, we have decided to implement the Random Forests machine learning algorithm in the MyriaL relational algebra. The Random Forests algorithm is an ensemble learning method, based on the construction of multiple decision trees from subsets of training data. Basic decision tree systems often suffer from data overfitting. Developed by Leo Breiman and Adele Cutler, Random Forests addresses this issue by selecting decision features randomly, as opposed to using techniques from information gain.

2. RELATED WORK

In [1], Leo Breiman presents the original motivation and formalization of random forests as an algorithm. He discusses their merits relative to contemporary alternatives, and details the algorithm, including discussing its usage for regression and classification. In [2], the authors discuss and demonstrate several aspects of the Myria system. Covered features include the interface and methods of interaction, the engine and associated algorithms, query languages, and service oriented aspects of the system.

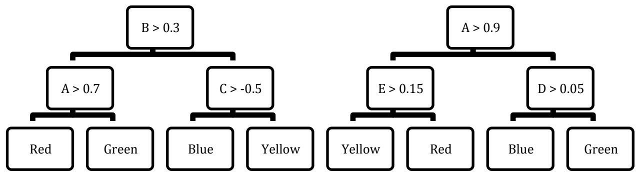


Figure 1. A contrived example of random feature selection influencing tree development. Of the top two most discriminating factors, 'A' is more so than 'B.' Consider if the tree selects 'B,' 'C,' and 'D' as features, as opposed to selecting 'A,' 'B,' and 'E.'

Additionally, they present the motivation and goals in developing the system. In [3], Liaw and Wiener discuss the R language interface to the initial Fortran implementation provided by Breiman and Cutler. They detail examples, as well as covering tuning. Additionally, they explore a variety of notes regarding practical usage of the system.

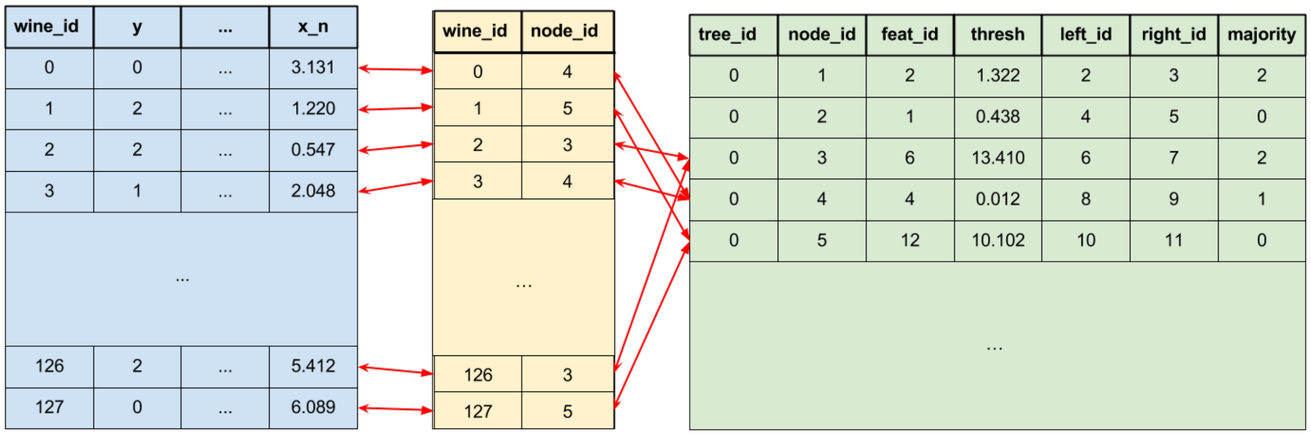
3. APPROACH

When approaching this project, we broke it into three main parts. First, we needed to select an appropriate dataset, that we were preferably interested in. Once we selected a dataset/as we selected, we could investigate potential questions, and choose a machine learning algorithm that would enable us to answer them effectively. We selected a smaller dataset, since learning over random trees can be computationally taxing. For example, while there are only 178 instances in the set we chose, there are thirteen attributes, allowing for large trees if desired. However, this dataset, and the potential questions, do fit well with random forests. They involve classification of instances into a small number of categories. Many other algorithms, including basic decision trees would likely overfit, a problem which random forests can address.

Second, we needed to familiarize ourselves with Myria and MyriaL, so that we could understand how to write our algorithm in the language, and run it on a shared-nothing cluster. This included gaining a basic understanding of the system architecture, as well as establishing local clusters on which to develop, alongside using their public demo for basic testing. Following this, we focused on learning and understanding the features and limitations of the language. We will address the difficulties we had with this later in the paper. Ultimately, however, we managed to cobble together an understanding of the language using a variety of examples, test code, the parser code, and experimentation.

The next step was the implementation of our algorithm. Although our implementation was focused around the selected dataset, we wanted to make it as general as possible. This included allowing tuning of a variety of parameters, such as the number of trees, the consensus percentage, and the number of features considered at each split. After establishing a basic overview of our sections, and agreeing on a variety of decisions, one researcher began work on tree building/splitting code, and the other started with code to evaluate and test the built trees, as well as produce predictions for unknown samples. After development of the core bits of both completed, work on connecting and fleshing out the modules commenced.

To test our implementation of random forests, and its performance, we followed the typical machine learning pattern of splitting our data into a separate training and testing set. Ultimately, while training and testing, we will want to keep track of a variety of metrics, including, for example, total time to build our model, model size, and time for a single evaluation, as well as the accuracy of our random forests model. Tracking all of these over several runs of the complete system will facilitate basic evaluation of the efficacy of our implementation. Following successful basic evaluation using the selected basic dataset, we will then be able to better generalize and set it up for use with larger datasets, and building a more complex model to more rigorously test our implementation.



wine_id	y	...	x_n
0	0	...	3.131
1	2	...	1.220
2	2	...	0.547
3	1	...	2.048
...			
126	2	...	5.412
127	0	...	6.089

wine_id	node_id
0	4
1	5
2	3
3	4
...	
126	3
127	5

tree_id	node_id	feat_id	thresh	left_id	right_id	majority
0	1	2	1.322	2	3	2
0	2	1	0.438	4	5	0
0	3	6	13.410	6	7	2
0	4	4	0.012	8	9	1
0	5	12	10.102	10	11	0
...						

Figure 2. ***Left:*** **The data in normalized format. Actually stored denormalized until dynamic column names become supported by MyriaL.** ***Center:*** **Our during training mapping from samples to their current split for the current tree.** ***Right:*** **A hypothetical truncated final tree mapping schema, demonstrating the information stored for later evaluation.**

4. ACCOMPLISHMENTS

First we focused on familiarizing ourselves with the Myria ecosystem in general. The UW Databases group maintains a demo cloud interface that allows basic query experimentation on a small cluster. We first started with this system, exploring the query language. Our next main task was getting it set up under our own control, learning how to instantiate clusters both locally and in AWS. From here we were able to further experiment with the system, practicing loading data and running queries. Among the example code we focused on, there exist a number of scripts, including some for PageRank and K-Means. We used these as a basic guideline in planning our own implementation.

Parallel to this, we also searched to find an appropriate dataset. While random forests can be adapted to work with a variety of problems, one of the simplest is basic classification. As our focus in this research project is on our MyriaL implementation of random forests, we chose a simple dataset, with a limited (but diverse enough) set of features. In this dataset, wines from a similar region in Tuscany are categorized as belonging to one of three cultivars. We also investigated several larger and more complex datasets, with the eye of making our evaluation more rigorous by utilizing an additional one.

Also in parallel, we investigated and dug into the theory and practice of the random forest algorithm, along with variations and related concepts. This included gaining a basic understanding of other implementations, which can help guide our version, both in standardization and convention. Following this, we discussed some details of our planned implementation, including our basic approach. We also considered several options for collaboration and development, including the usage of git for versioning and coordination.

In the end we split much of the code work. Some of this was planned, and some happened organically. For the most part, one of us worked on getting the decision tree code written while the other worked to split the data and write the random forest algorithm. During this time, we also worked to write this paper.

5. Development

Our development process went in a few stages, with most stages ending in a serious rollback/modification based on newfound Myria limitations.

At first we focused on a recursive implementation, that kept several tables, one for each node, with a split simply operating on a given table, and forwarding each half to be split again as needed. This initial implementation mostly demonstrated our instinctual object-oriented approach. This of course did not take near full advantage of Myria as a language, and in fact would likely be counter-intuitive to a database systems engineer. In addition, we were hoping for a median calculation for our split. What we didn't initially realize is that in a shared-nothing language, a median calculation would be slow and difficult to implement.

After discussing our project with a Myria researcher, we discovered the need to shift our implementation for a couple of reasons. For one, Myria does not allow for recursion. For another, it was clear that it would make much more sense to keep one table mapping data to an associated node, and remap every time a split is made. This was a much more elegant approach, taking full advantage of Myria temporary tables and easy updates/unions. We also made a few small changes, adding some columns to our final decision tree table, which in the end was {tree_id, node_id, feature_id, threshold, left_id, right_id, majority}. We also decided initially to use a mean calculation for the threshold, which although is not in the true spirit of decision trees, allowed for quicker iteration of code. As we had local setups, without a huge cluster to test on initially, we developed and tested individual sections of our code as we wrote. Once we had all the desired pieces, we worked to connect them. After some syntax testing, and lots of careful checking of our code, it seemed that we were ready to attempt a run. We had our decision tree code, and code to build a forest from this, with a few adjustable parameters, and additionally had our data in the Myria system in the expected denormalized format.

tree_id	*node_id*	*feature_id*	*threshold*	*left_id*	*right_id*	*majority*
0	1	1	0.3	2	3	0
0	2	0	0.7	4	5	0
0	3	2	-0.5	6	7	3
0	4	-1	-1	-1	-1	0
0	5	-1	-1	-1	-1	1
0	6	-1	-1	-1	-1	2
0	7	-1	-1	-1	-1	3

Figure 3. The left tree from figure 1. recorded as it would be under our final tree/split schema.

When we attempted to combine everything, however, we found more limitations of the Myria system, including that our knowledge of functions was imperfect, at best. After some more digging, we moved away from complex functions, due to apparent limitations on functionality, planning to potentially refactor back to functions after getting code running. In an attempt to get our algorithm working, we decided to unroll our methods, and keep our training code contained in a single script. We kept liberal documentation in an attempt to maintain code readability. Additionally, we reworked some of our code at this point, to match better with typical implementations for random forests. For example, instead of using the mean of values for a given feature as the threshold, we switched to calculating the optimal threshold from a number of candidates (generated as evenly spaced intervals along the range of values), using information gain.

After making these modifications, we again attempted to run our code on the Myria system. We came across a number of errors, however, ultimately suggesting a lack of support for nested loops. This was confirmed by several Myria/MyriaL researchers, who also suggested that modifications to support this would be difficult. Without recursion, nested loops, or a few other missing features, a random forest implementation appears nigh impossible.

Not wanting to give up, however, we delved back into the parser and interpreter code, searching for a way to support nested loops. Working with a basic nested loop variable-incrementation example, we managed to enable the system to parse nested loops, and fix a few locations where it could not handle their execution. Moving back to our code, however, our lack of a holistic understanding of the Myria(L) system proved problematic, as it appeared the greater complexity of random forests caused unforeseen complications, preventing full execution, and stymieing our efforts. The last remaining option, hard-coding iterations, defeats the entire purpose of the project, to build a generalized, tunable random forests implementation in Myria, unfortunately. We plan to try this option shortly, in order to demonstrate the correctness of our code, but do not believe any numbers produced by such a process to be indicative of overall performance. Nor will this reflect the general versatility of our random forests implementation. As such, we focused on additions to Myria(L) as a higher priority contribution over this option. Towards this end, both to improve our code, as well as to add to the system, we dove back into the Myria code, and implemented the ability to generate sequences in an efficient parallelized fashion.

6. WORKING WITH MYRIA(L)

We had never worked with Myria, so as expected most of the difficulty lay in learning the language and shifting the algorithm from our natural object-oriented approach into a new paradigm. Once we managed to shift our thinking, Myria enabled us to do lots of powerful work. For example, splitting our decision trees, we found the easiest way was not keeping separate structures for the samples belonging to each node, as an object oriented approach would suggest, but instead maintaining a single table, mapping node ids to data ids. We could then just remap, which was quite space efficient and saved overhead.

In general, the language seems powerful. It easily allowed us to query data across a distributed system, with the actual nuances of data distribution remaining hidden. Additionally, the shared university connection added to its allure. Overall, it seems to be a very well put together language, once you adjust to their mix of an imperative, relational algebra paradigm.

We did however, have a number of challenges with the language, which we believe to be weak, but addressable points of the language. Perhaps the most frustrating thing about it was the lack of documentation. MyriaL appears largely undocumented. The closest thing we found to references were a few locations with examples, documented themselves primarily only through inline comments, and which covered only a small subset of the language features. For someone new to the language, this was extremely frustrating. We cobbled together our understanding from these, alongside a few test examples we managed to find, and by reading through the parsing code. There are many pieces of syntax that were difficult to understand. We also never got a good grasp on how to write functions, and the syntax for many calls, nor the limitations of them. Granted the language may have not been built to be the first shared-nothing database language people are learning, but we found it very difficult to learn both the shared-nothing paradigm and Myria at the same time. After a quarter of working with it, we do think MyriaL has potential as a powerful language (with some feature iteration), but in order to gain any degree of adoption among the developer community, it will require more comprehensive documentation.

Towards the end of this project we also dealt with another problem. We found that methods were extremely delicate and limited. With our limited understanding of their syntax and functionality, we eschewed them in favor of a less modular, more repeated script.

Implementing random forests, however, with any degree of customizability to fit arbitrary data requires nested do-while loop, with dynamic iteration lengths. However, when connecting the individually developed portions of our code, we discovered that MyriaL does not currently support nested do-while loops. In an attempt to address this, we dove into the language parser and interpreter. With some work, we were able to get basic examples of nested loops working. However, the random forests implementation relies on more complex functionality, and without a good understanding of the language and system as a whole we were unable to patch it to support our code though execution. In order to improve our code, as well as contribute more directly to the Myria project, we also worked in the MyriaL parser/interpreter and MyriaX to implement a sequence generation mechanism. The resulting operator allows robust generation of specific sized relations, in a full implicitly parallelized fashion.

7. CONTRIBUTIONS

Our achievements and contributions include the following:

- Theoretical implementation of random forests in MyriaL, awaiting a more robust implementation of nested looping in order to execute correctly.
- Additional MyriaL documentation through liberal commenting of our code
- Basic implementation of nested do/while looping in MyriaL. This works for simple loops, and can serve as a starting point for a more robust implementation.
- Full implementation of parallelized sequence generation in MyriaL and MyriaX. For this. added `seq(NUM)` to generate a sequence from 0 to NUM minus 1.
- This collection of experiences and feedback from developers beginning on fresh on the system, as well as code produced by such 'novice' MyriaL programmers.

As part of this submission, we will be sending this report and our code to the Myria researchers. This includes our forks of Myria and RACO, as well as the repository containing our collected random forests code (Appendix A). We hope that our feedback proves valuable in guiding future development and documentation, and that our code proves useful for both, as well. Despite the inability to run our random forests implementation with the current Myria system, further feature development should unlock its functionality, and it can also stand as an example of a collection of code snippets for other developers on its own. Additionally, it provides a window into how 'novice' Myria developers approach the system, and begin writing code. We hope that this, combined with our other concrete contributions will prove valuable and useful for future Myria development.

8. DIVISION OF LABOR

Both of us worked on general understanding of the system architecture and MyriaL itself, passing back and forth snippets of code, and key files to inspect. Both of us participated in planning our code architecture. During initial coding, Connor focused on handling a given split, while Nick worked on setting up a testing framework. Following this, Nick handled compiling and connecting the code modules initially, as well as converting to a single nested-loop script later, while Connor researched a number of the language limitations, and worked on generating this report. Connor continued these tasks, and Nick worked on the Myria parser/interpreter, attempting to add nested-loop functionality, and then worked on designing the poster. Both of us worked on editing both the poster and the paper, as well as planning out general content, and attempting to brainstorm solutions and ways to address the MyriaL limitations. Nick added sequence generation to MyriaL and the Myria execution system.

9. ACKNOWLEDGEMENTS

We would like to thank a number of Myria researchers and engineers for their assistance in various phases of the project. Additionally, we would like to thank the teaching staff of CSE 544, for their assistance and feedback, as well as teaching us this quarter. Beyond that, we would like to thank the general Myria team as well, for their work on the system, and express our hope that they find this a valuable and useful contribution to future development. Particular thanks to: Tobin Baker, Alvin Cheung, Shumo Chu, Brandon Haynes, and Brandon Myers.

10. REFERENCES

[1] Breiman, L. (2001). Random Forests (R. Schapire, Ed.). *Machine Learning, 45*, 5-32.
[2] Halperin, D., Whitaker, A., Xu, S., Balazinska, M., Howe, B., Suciu, D., . . . Wang, J. (n.d.). Demonstration of the Myria big data management service. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data - SIGMOD '14*.
[3] Liaw, A., & Wiener, M. (2002). Classification and Regression by randomForest. *R News, 2*(3), 18-22. doi:ISSN 1609-3631

A. APPENDIX

Here we have collected links to various code repositories:

- Our fork of RACO: https://github.com/scriptreiter/raco
- Our fork of Myria: https://github.com/scriptreiter/myria
- Our random forests code: https://github.com/scriptreiter/myria-forests (Note: this requires a cse or uw netid. For access otherwise, contact the authors.)